PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 :
G06F 9/445, 9/45

A1

(11) International Publication Number: WO 96/12224

(43) International Publication Date: 25 April 1996 (25.04.96)

(21) International Application Number: PCT/US95/06059

(22) International Filing Date: 12 May 1995 (12.05.95)

(30) Priority Data:
08/324,810 18 October 1994 (18.10.94) US

(71) Applicant: MARCAM CORPORATION [US/US]; 95 Wells Avenue, Newton, MA 02159 (US).

(72) Inventors: REED, Harvey, G.; 7 Royal Crest Drive #1, Marlborough, MA 01752 (US). DALTON, John, T.; 166 Hunt Road, Chelmsford, MA 01824 (US). AIBEL, Jonathan, B.; 574 Harrington Avenue, Concord, MA 01752 (US). SCIANDRA, Stephen, A.; 34 Newcombs Mill Road, Kingston, MA 02364 (US).

(74) Agents: POWSNER, David, J. et al.; Choate, Hall & Stewart, Exchange Place, 53 State Street, Boston, MA 02109-2891 (US).

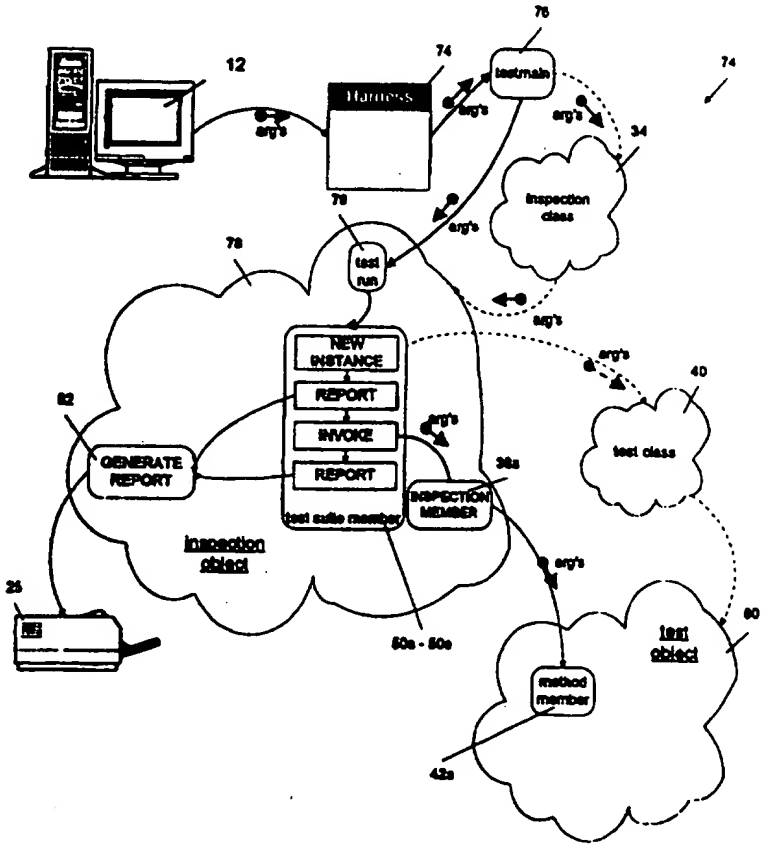
(81) Designated States: CA, JP, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.

(54) Title: METHOD AND APPARATUS FOR TESTING OBJECT-ORIENTED PROGRAMMING CONSTRUCTS

(57) Abstract

The invention provides methods and apparatus for generating, from a source signal defining a subject class to be tested (40), an inspection signal defining an inspection class (34) that has one or more members for: (i) generating a test object as an instantiation of the subject class or a class derived therefrom, (ii) invoking one or more of the selected method members of the test object, and (iii) generating a reporting signal based on an outcome of invocation of those members. The inspection class, as defined by the inspection signal, can include one or more method members (50f), referred to as "inspection members", for testing corresponding method members of the test object (and, therefore, in the subject class). So-called "test suite members" (50a-50e), that are also defined as part of the inspection class, exercise the inspection members. The invention also provides methods and apparatus for responding to an inspection signal to create an inspection object instantiating the inspection class. Members of the inspection object are invoked to create the test object (80), to invoke method members thereof and to generate a signal reporting an effect of such invocation. The test object members can be invoked by corresponding inspection members of the inspection class, which in turn can be invoked by test suite members of the inspection class.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria
AU	Australia
BB	Barbados
BE	Belgium
BF	Burkina Faso
BG	Bulgaria
BJ	Benin
BR	Brazil
BY	Belarus
CA	Canada
CF	Central African Republic
CG	Congo
CH	Switzerland
CI	Côte d'Ivoire
CM	Cameroon
CN	China
CS	Czechoslovakia
CZ	Czech Republic
DE	Germany
DK	Denmark
ES	Spain
FI	Finland
FR	France
GA	Gabon
GB	United Kingdom
GE	Georgia
GN	Guinea
GR	Greece
HU	Hungary
IE	Ireland
IT	Italy
JP	Japan
KE	Kenya
KG	Kyrgystan
KP	Democratic People's Republic of Korea
KR	Republic of Korea
KZ	Kazakhstan
LI	Liechtenstein
LK	Sri Lanka
LU	Luxembourg
LV	Latvia
MC	Monaco
MD	Republic of Moldova
MG	Madagascar
ML	Mali
MN	Mongolia
MR	Mauritania
MW	Malawi
NE	Niger
NL	Netherlands
NO	Norway
NZ	New Zealand
PL	Poland
PT	Portugal
RO	Romania
RU	Russian Federation
SD	Sudan
SE	Sweden
SI	Slovenia
SK	Slovakia
SN	Senegal
TD	Chad
TG	Togo
TJ	Tajikistan
TT	Trinidad and Tobago
UA	Ukraine
US	United States of America
UZ	Uzbekistan
VN	Viet Nam

METHOD AND APPARATUS FOR TESTING OBJECT-ORIENTED PROGRAMMING CONSTRUCTS

BACKGROUND OF THE INVENTION

The invention described herein pertains to digital data processing and, more particularly, to methods and apparatus for testing object-oriented programming constructs.

There are many competing factors determining how an organization tests its software. In the commercial environment, this includes allocating development resources, motivating developers to write unit tests, and maintaining those tests over the lifetime of the product. The central problem is focusing limited development resources on the unique aspects of unit tests, minimizing the overhead of creating test environments and packaging. Secondary factors include choosing which techniques (state-based, boundary analysis, etc.) are applicable to validate each production unit.

Traditional unit testing validates the function points within a software product to verify the correctness of its operation. Often, the strategy is to write a dedicated test program for each unit, requiring separate test control and execution, set up of the test environment, and production of output to describe the execution of the test, as shown in **FIGURE 1**.

Implementing this strategy is often informal and ad hoc. Frequently unit testing is abandoned because of the difficulty of this approach. When a developer determines that some code does need unit testing, he will create his own test harness, writing code to set up the test environment, run the test case, output the results and shutdown the test environment. The developer is concerned about all support issues, such as memory management, output device control and formatting, and initializing and cleaning up the application-specific environment. The actual test code is generally a small percentage of the entire unit test.

This practice also tends to be invasive, where conditional test code is added to the production code to aid in debugging. Typically this code consists of commands to output the value of variables, trace the execution of the code and perhaps validate the state of the

environment. Conditionally compiled code introduces a new dimension of complexity into the production code and bugs can be introduced that only appear when this conditional code is disabled. Thielen, No Bugs! Delivering Error-Free Code in C and C++, Addison Wesley Publishing Co. (1992) notes that this is a common problem.

One way to save time is to cut and paste code from previous test cases. The cut and paste development methodology saves time in developing the code, but it also requires that changes be propagated to multiple places in the test code, increasing maintenance cost. The amount of time which is saved by this method is typically only short term. Since the test code is usually not part of a developer's deliverables, code standards may not be followed and often each developer creates his or her own set of tools for managing the environment and tests. This makes it hard for developers to share unit test code outside of a development group, for example in quality assurance (QA) or system integration groups. Moreover, if the test code is not included in the source control system, it may become lost or thrown away when it becomes obsolete.

Some development organizations, when faced with the expense of developing unit test code or with the expense of maintaining an unwieldy collection of disorganized test programs, give up on the test effort, leaving validation to the QA group. This is particularly detrimental in an object-oriented programming environment where a system test approach is less effective in detecting certain classes of errors than a unit test approach.

Programs produced using procedural techniques are highly function oriented, where programs produced with object-oriented techniques have a much greater emphasis upon the data being manipulated. This places a different emphasis on the program's construction. Traditional testing techniques can be adapted to object-oriented programs, however, they are not guaranteed to exercise all of the facilities that are peculiar to object-oriented programming.

Traditional practice in industry is to continue using the procedural organization of unit tests for class-based production code, as shown in **FIGURE 2**. Since the number of classes in commercial applications can be quite large, the cut and paste technique is even more costly. Most important is the paradigm mismatch between class-based production code and procedural unit testing.

Objects consist of both a state and a defined behavior (operations). The functionality of the class is defined by both the operations and the interaction between the operations and the state. Traditional testing techniques can eventually exercise many of these interactions however, this is not guaranteed. Additionally, object-oriented programming uses techniques such as polymorphism which are rarely found in traditional procedural programs.

Current research into the testing of object-oriented programs is concerned with a number of issues. The black-box nature of classes and objects implies that a black-box testing technique would be beneficial. A number of black-box approaches have been developed, including the module validation technique of Hoffman *et al* "Graph-Based Class Testing," Proceedings of the 7th Australian Software Engineering Conference, ASWEC (1993) and "Graph-Based Module Testing," Proceedings of the 16th Australian Computer Science Conference, pp. 479 - 487, Australian Computer Science Communications, Queensland University of Technology, and the ASTOOT suite of tools by Frankl *et al*, "Testing Object-Oriented Programs," Proceedings of the 8th Pacific Northwest Conference on Software Quality, pp. 309 - 324 (1990). Both of these techniques concentrate upon the modular nature of object oriented programs. However Fielder, "Object-Oriented Unit Testing," Hewlett-Packard Journal, pp. 69 74, April (1989) notes that a more adequate testing technique can be gained by combining black-box testing with the use of a whitebox coverage measure.

In view of the foregoing, an object of the invention is to provide improved methods and apparatus for testing object-oriented programming constructs. More particularly, an object is to provide such methods and apparatus for testing classes in object-oriented programs and libraries.

A related object is to provide such methods and apparatus for testing all members of a class and exercising each member over a full range of expected runtime parameters.

Still another object is to provide such methods and apparatus for implementation in a wide range of digital data processing operating environments.

SUMMARY OF THE INVENTION

The invention provides methods and apparatus for testing object-oriented software systems and, more particularly, class constructs that provide the basis for programming and data structures used in those systems.

In one broad aspect, a method according to the invention calls for generating, from a source signal defining a subject class to be tested, an inspection signal defining an inspection class that has one or more members for (i) generating a test object as an instantiation of the subject class or a class derived therefrom, (ii) invoking one or more selected method members of the test object, and (iii) generating a reporting signal based upon an outcome of invocation of those members. The source and inspection signals can be, for example, digital representations of source-code programming instructions of the type typically contained in source-code "header" files.

The inspection class, as defined by the inspection signal, can include one or more method members, referred to as "inspection members," for testing corresponding method members of the test object and, therefore, of the subject class. Thus, for example, the inspection class can include inspection members corresponding to, and taking similar arguments to, constructors and operators in the test object. Likewise, the inspection class ca include members corresponding to destructors in the test object. The inspection members c have function names similar to those of their corresponding method members of the test object.

So-called test suite members, that are also defined as part of the inspection class, car invoke for test purposes, or exercise, the inspection members. The test suite members can, for example, test accessor, transformer, operator or semantically unique members of the test object. The test suite members can also test the persistence of the test object, as well as memory leaks associated with its creation or destruction. A "run test" member can also be included in the inspection class in order to invoke the test suite.

According to further aspects of the invention, the inspection class includes method members that provide common services used by the test suite members. These include reporting services that permit uniform generation of tracking and error reports.

Methods as described above can further include generating, from the source signal, a test signal that defines a test class. The test class may comprise the subject class, or an instantiable class derived therefrom, that inherits members of the subject class. In related aspects of the invention, the test class substantially duplicates pure virtual functions of the subject class, absent programming constructs that denote those functions as having both pu and virtual attributes (e.g., the "pure" and "virtual" keywords). The test class can also be defined to give instantiations of the inspection class access to members of the subject class, e.g., via so-called "friend" declarations. In accord with these methods, the inspection class includes members that create the test object as an instantiation of the test class.

The invention provides, in yet another aspect, methodology that responds to an inspection signal to create an inspection object instantiating the inspection class. Members the inspection object are invoked to create the test object, to invoke method members there and to generate a signal reporting an effect of such invocation. The test object members ca be invoked by corresponding inspection members of the inspection class, which in turn can be invoked by test suite members of the inspection class, as described above.

According to other aspects of the invention, arguments are applied to the test object upon instantiation of the object, (in the case of constructors) or upon invocation of the

selected member methods. Those arguments can, by way of example, be specified interactively or generated automatically.

In a related aspect of the invention, invocation of a member function is reported by comparing the results of invocation of a member function with an expected value. In the case of disagreement, an error message can be displayed, e.g., to the user console or to an output file. By way of further example, the contents of the test object can be "dumped" following invocation. Moreover, signals indicative of the results of these and other results of invocation, e.g., statistics regarding the number of successful and erred member function calls, can be stored in data members of the inspection object. Reporting signals, such as the foregoing, generated in accord with the invention can be generated at selected verbosity levels.

Still other aspects of the invention provide an apparatus for testing object-oriented software systems having functionality for carrying out the functions described above.

These and other aspects of the invention are evident in the description that follows and in the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

A more complete understanding of the invention may be attained by reference to the drawings, in which:

Figure 1 depicts a prior art strategy for unit testing of procedural code;

Figure 2 depicts a prior art strategy for testing object-oriented code;

Figure 3 depicts a preferred computing system for use in practice of the invention;

Figure 4 depicts a production class and an inspection class according to the invention for testing thereof;

Figure 5 depicts a relationship between the inspection class and test monitor and test kernel superclasses, as well as between the test subclass, the production class and the production superclass;

Figure 6 depicts a test suite and inspection methods comprising an inspection class according to the invention;

Figure 7 depicts illustrative states an object can enter under the persistence paradigm;

Figure 8 depicts a relationship of persistence testing methods in an inspection class according to the invention;

Figure 9 depicts operation of an inspection class generator according to the inventio and

Figure 10 depicts operation of a test harness according to the invention;

Figure 11 depicts a window displayed and used in a test harness according to the invention.

DETAILED DESCRIPTION OF THE ILLUSTRATED EMBODIMENT

FIGURE 3 depicts a preferred networked computing system for use in operation of the invention. The system includes a digital data processor 12, including a processor sectio 14, a random access memory section 16 and an input/output control section 18. The digital data processor 12 is connected, via input/output control section 18, to monitor 24 (includin monitor, keyboard and pointing device) and to hard disk 22 for storage of software and data Digital data processor 12, as well as its sub-components 14 - 18 and peripherals 22 - 24, preferably comprise a conventional commercially available personal computer or work station adapted in accord with the teachings below for the testing of object-oriented softwar As further indicated by the drawing, digital data processor 12 is connected with printer 25 and with other other digital data processing apparatus 27A, 27B, 27C via network 26.

INSPECTION METHODS

Reed, "Object-Oriented Design by Orthogonality," Computer Language, vol. 9, no pp. 51 - 56 (1992), discloses an object-oriented library architecture using orthogonality Building on that architecture, methods and apparatus according to the invention utilize inspection methods to package test code and inspection classes package inspection method

Referring to **FIGURE 4**, a production class 30 (i.e., a class from the production cod to be tested includes three method members A (element 32a), B (element 32b), and C (element 32c). Method member A, by way of example, is a Boolean function that takes an integer argument and is prototyped, e.g., in the production class header file, as "bool A(int)

An inspection class 34 according to the invention includes three method members I_Class::A (element 36a), I_Class::B (element 36b), I_Class::C (element 36c) correspondi to like-named method members A, B and C, respectively, of production class 30. Class 34 defined as part 38a of unit test source code stored in a conventional manner (e.g., on disk 2 in source and header files 38 for compilation, linking, loading and execution using

commercially operating system tools in connection with libraries constructed in accord with the teachings herein.

As shown in the drawing, method member I_Class::A takes arguments and returns values identical to those of corresponding member A. Member I_Class::A takes, as an additional argument, a pointer to a test object created from the production class or a test class derived therefrom. As further shown in the drawing, method I_Class::A includes instructions for invoking member A of the test object, as well as for reporting invocation, arguments to, and return values from that member. Thus, by way of example, where member A is declared as follows:

```
bool A(Int      arg1);
```

I_Class::A can be declared as follows:

```
bool
I_CLASS::Class:: A(P_Class*  Obj,
                   Int       arg1)
{
  AnnounceStart("A");              // report start of test
  AnnounceParameter(arg1);         // report parameters that will be
                                   // passed to member to be tested
  bool tstVal = Obj->A(arg1);      // invoke member
  AnnounceRtnValue(tstVal);        // report member result
  AnnounceEnd("A");                // report end of test
  return(tstval);                  // return member result
}
```

Each of the inspection methods take the object under test as a parameter, as well as the rest of the parameters necessary for the production method, as shown in **FIGURE 4**. This not only reduces cut and paste code, but also eliminates the need for specialized test code in the production class that is used only during unit testing. It is much preferred to have instrumentation outside of production code rather than compiled in, even if the compilation is conditional.

The inspection class methods are capable of running against code built for general release as well as development and unit testing. With a "dump" method enabled during development, unit tests facilitated by the inspection class provide a white-box testing environment, allowing the developer to be most effective in determining corrective actions for failed tests. Black-box test techniques are in effect when the dump method is disabled for

general release. Unit tests in this form can be run as acceptance tests by QA prior to produc level tests. The dump method is a member method preferably included in all production classes that prints, or "dumps," state information of the contained member data and objects.

INSPECTION CLASSES

From the inspection methods, an "inspection class" providing a non-invasive instrumented interface is constructed. This interface includes an inspection method for each public and protected member in the production class. The inspection class interface is analogous to the production class interface; thus, developers need to learn very little in orde to construct unit tests.

The inspection class 34 and its inspection methods 36a - 36c are can be generated by parsing the production header files and constructing inspection class headers and source file from the production code header files.

THE TEST SUBCLASS

Referring to **FIGURE 5**, there is shown a test subclass 40 that derives from production class 30. Thus, test subclass 40 includes method members 42a, 42b, 42c corresponding to method members 32a, 32b, 32c, respectively of the production class 30.

Definition of a test subclass 40 according to the invention solves problems associate with accessing protected production methods or instantiating abstract classes. Since the test subclass 40 is derived from the production class 30, it has access to the protected methods o its parent. Access to such protected methods by the inspection class 34 is granted by inclusion, in the test subclass definition, of statements declaring the inspection class as a "friend" (in C++ terminology). This insures that testing is completely non-invasive. For those classes which do not need a test class, the test class 40 is replaced with an alias to the actual production class 30.

The test subclass also provides a location for the implementation of the pure virtual functions of the production class. Typically, those functions are implemented by stubbed-o code.

As further shown in **FIGURE 5**, the production class derives from a production superclass 44, while the inspection class 34 derives from a test monitor superclass class 46 which, in turn, derives from a test kernel superclass 48. The test monitor class 46 defines data members for storing results of each unit test. The test kernel class 48 defines method members for uniform reporting of unit test results, status and coverage.

Given a set of inspection methods 36a - 36c grouped into an inspection class 34, common services of test output and test environment maintenance are provided by test instrumentation base classes 46, 48. The kernel test class 48 allows infrastructure objects to be tested in isolation. It provides services for output verbosity control, output formatting, memory management leak detection and reporting, expression checking, and reporting of errors encountered.

The monitor test class 46, derived from the kernel test class 48, provides additional services to application classes such as application environment initialization and cleanup, test case setup and cleanup, gross level coverage measurement and reporting, and unit test execution.

ORGANIZATION OF TEST SUITES

Source code and header files providing definitions of the inspection class 34 and test subclass 40 are preferably generated automatically from a header file defining the production class. This results in a reduction in the amount of code the developer needs to write in order to insure thorough testing of the production class. This section examines the definition and interrelationship of unit tests within the inspection class.

Unit tests must achieve a certain minimum standard of coverage. This is ensured by dividing the set of production class methods, e.g., 32a - 32c, into groups according to behavior and testing technique, taking into account dependencies between various groups. For example, all methods depend on the constructor methods. Referring to **FIGURE 6**, production class 30 methods are tested by the following groups of inspection class tests: memory lifecycle 50a, persistence lifecycle 50b, get and set attributes 50c, operator members 50d, and semantics 50e.

Memory lifecycle tests 50a exercise those methods of the production class which affect the life of the corresponding object in memory. These include the constructors, copy constructors, destructor and assignment methods of the production class. Generally, errors which occur in the lifecycle are memory leaks and improper initialization. Lifecycle tests involve creating test objects (based on the production class, or the test class derived therefrom) using each constructor in the production class 30, assigning objects (e.g., in the case of copy constructors), and deleting the test objects from memory. Checks are then made for leftover allocated memory.

Persistence lifecycle tests 50b exercise those objects that are stored in a database (e.g., on disk 22) for later retrieval or storage across invocations of the product. A state-based testing approach is used for persistent methods. The states defined in the memory and

persistence lifecycles are so regular that a majority of the test cases in these sections can be predefined and therefore implemented by inspection class generator apparatus and methods according to the invention, thereby reducing developer effort.

Get and set tests 50c exercise those methods which allow the access to and setting of value-based attributes of the production class. Testing assures that the values can be accessed, and that ancillary properties such as "read-only" are complied with. A state-based testing approach is preferably used for this group.

Operator tests exercise 50d those methods of the production class that perform comparison, arithmetic, logical operations and others. Operator testing is functional in natu and relies on techniques such as equivalence class partitioning and boundary value analysis.

Semantics tests exercise 50e those members of the production class whose behavior that makes the object unique. Because of that uniqueness the inspection class generator apparatus and methods do not attempt to generate code for exercising those members but, rather, they merely generate method declarations (and comments) to remind the developer t draft his or her own test code.

TEST SUITE PACKAGING

An inspection class generator generatesindividual tests, along with inspection methods 50f providing the instrumented interface. These inspection methods are, themselv packaged into "test suites" As a consequence of the resulting uniform organization of all inspection classes, the set of unit test suites can be executed in a uniform manner.

A standard entry point, e.g., referred to as "testmain()." in testing code for constructing an instance of the inspection class and invoking a method member, e.g., "testrun()," that exercises the members of the test suite. The implementation of testmain() i a nonclass function which is automatically generated by the inspection class generator.

PERSISTENCE TESTING

The following sections examine the application of state-based testing to the persistence of objects. State-based testing and its applicability are described. An example a multi-state object is shown. Some additional testing problems due to the specific persistence approach are also discussed.

STATE BASED TESTING

Of the various methods considered in developing test cases, state-based testing was the most applicable to persistence. State-based testing was developed at the University of

Durham to address some of the problems in adapting testing techniques to object-oriented programming, and is described in Turner *et al*, "The Testing of Object-Oriented Programs," Technical Report TR-13/92, University of Durham, England (1992).

An object's behavior is defined by the code within its operations. Events and situations occur that have to be communicated to the other operations of the object. This communication is usually performed by storing specific values within the attributes of the class. State-based testing models these interactions as a finite-state-automata which is then used to predict the response of each operation to a particular state.

Test cases are generated to validate that at no point in an object' s lifetime can it be placed into an undefined or incorrect state. All operations are validated for the state changes which they are defined for, operations can also be validated for their robustness when faced with an illegal state situation.

An example deals with persistent classes. Persistence is implemented by a root persistent class from which all persistent objects are derived. The persistence paradigm requires six states that an object can enter:

New - the object was created and exists in memory only

Saved - the object exists in the database and not in memory

Open for Read/Write - the object exists (and is identical) in memory and the database and the client may save changes to the object

Modified Read/Write - the object exists in memory and the database with different values and the client may make changes to the object

Open for Read Only - the object exists (and is identical) in memory and the database and the client may not make changes to the object

Deleted - the object does not exist in memory or the database

FIGURE 7 illustrates some of the legal transitions between these states. Thus, for example, an object may transition from a New state 52a, or a Modified Read/Write state 52e, into a Saved state 52b. From that state 52b, an object may transition to an Open Read Only 52c or an Open Read/Write state 52d, whence it may transition to a Deleted state 52f. An object may also transition into a Deleted state from the Modified Read/Write state 52e.

Since the root persistent class is inherited by all persistent classes, these classes share the above states 52a - 52f, although the implementation is made specific to each class by

providing some of the persistence code in a code template that is instantiated for each class. The test cases for these objects should also be similar, with minor changes from class to clas Because the persistence functions can be critical to the operation of an object-oriented product, it is not sufficient to test the functions solely at the root level.

The situation of shared critical functionality leads to another testing problem. Given the similarity of the tests, interest in rewriting the same test cases over and over for the persistent classes is likely to be low.

As illustrated in **FIGURE 8**, the inspection class generator conditionally generates the persistence lifecycle tests 50b to include standard persistence state tests 54. The condition which causes this generation is if the inspection class generator detects that the production class is persistent.

This solution allows each production class 30 to be tested with a minimum set of tes cases assuring a certain level of test coverage. The developer must supply a test setup routi 56 specific to the class under test. This is a short and well-defined piece of code, and the inspection class generator supplies a function body skeleton. Then the developer can code additional test cases as deemed necessary to the particulars of the class 30.

An additional advantage to this approach is that it provides a single point for the maintenance of the test cases. If the persistence application program interface (API) is modified, the amount of test code to be changed is minimized.

This does not appear to be an unique situation; it seems likely that there will be othe cases where a whole model - an API and series of states - is shared among many classes. Using state-based testing provides a significant savings here by allowing the development o a template of standard cases which are shared by the inspection classes. It should be noted that this approach not only demonstrates the correctness of the code under test but also demonstrates the correct integration of the shared production code template.

INSPECTION CODE GENERATOR

FIGURE 9 illustrates operation of an inspection code generator (ICG) 58 according to the invention. The generator 58 accepts, as input, a header file 60 defining the productio class 30 be tested. The generator 58 generates, in accord with the teachings herein, a heade file 62a and a source code file 62b defining the inspection class 34 and the test class 40 for testing the production class 30.

In the illustrated embodiment, files 60 and 62a - 62b contain C++ programming statements in source-code format. The header file 62a contains function declarations or

prototypes corresponding to those in the file 62b. Those skilled in the art will appreciate that the teachings herein are likewise applicable to other programming languages including object-oriented constructs, e.g., Turbo Pascal. Likewise, it will be appreciated that the information in the files may be input or output in other formats (e.g., object code) known in the art.

The statements contained in files 62a, 62b define the inspection class 34 to include members that (i) generate a test object as an instantiation of the test class, (ii) invoke selected method members of the test object, and (iii) generate a reporting signal based upon an outcome of invocation of those members.

The inspection class 34 particularly includes inspection members (also referred to as "inspection" members) that test corresponding method members of the test object. Thus, for example, the inspection class 34 includes inspection members, e.g., 36a - 36c, corresponding to, and taking similar arguments to, constructors, destructors, operators and other method members, e.g., 32a - 32c, in the test object. To facilitate testing, the inspection members (apart from constructors and destructors) have function names similar to those of the corresponding method members of the test object that they test.

The inspection class 34 defined by files 62a, 62b also includes test suite members that invoke or exercise the inspection members for test purposes. The test suite members 50a - 50e that test accessor, transformer, operator or semantically unique members of the test object, as well as the persistence of the test object and memory leaks associated with its creation or destruction. A "test main" member 50f invokes the test suite. The inspection class provides, via inheritance from the test monitor class and test kernel class 48, members 50f that permit uniform tracking and reporting of test coverage and errors.

The statements in files 62a, 62b also define a test class 40 that derives from the production class 30 and, therefore, inherits members, e.g., 32a - 32c, therefrom. The test class 40 duplicates pure virtual functions of the subject class 30, absent programming constructs that denote those functions as having both pure and virtual attributes (e.g., the "pure" and "virtual" keywords). The test class 40 affords the inspection class members, e.g., 36a - 36c, access to members, e.g., 32a - 32c, of the subject class, e.g., via C++ friend declarations.

Generation of the inspection class header and source files 62a, 62b by the ICG 58 may be understood by reference to the Appendix A, providing a listing of a sample production class header file 60 named "utg.hh"; Appendix B, providing a listing of a sample inspection class header file 62a named "i_utg.hh" that is generated by the ICG 58 from file 60; and,

Appendix C, providing a listing of a sample inspection class code file 62b named "i_utg.cpp that is also generated by the ICG 58 from file 60.

The items listed below refer to the annotations made on the listings provided by Appendices A - C. In the description of each item, reference is made to the three classes involved in the unit test:

1. The source production class 30, which may be an abstract or concrete class.

2. The test class 40. This is a generated class which is derived from the production clas to provide access to the public and protected member functions of the production class, and declarations for implementations of virtual functions declared pure virtual in the production class.

3. The inspection class 34. This class performs tests on instances of the test class 40.

Where dictated by context, references in the explanatory items below to the "production class" shall be interpreted as references to the corresponding file 60, and references to the "inspection class" and the "test class" shall be interpreted references to the files 62a, 62b.

Explanation of Annotations to Production Class
File 60 Listing of Appendix A

1. Copyright notice comment.

2. Include file guards are based on format <filename>HH.

3. Filename utg.hh becomes "i_utg.hh" for inspection class header file 62a. The inspection class source file 62b becomes "i_utg.cpp."

4. All #include's lines in production class file 60 are ignored.

5. Class comment is not copied to the inspection class files 62a, 62b.

6. The !LIBRARY keyword value (ER) is used for generating the inspection class file 62a, 62b comments.

7. The !NAME keyword value (utg) is used for generating the inspection class file 62a 62b comment.

8. The name of the production class (utg) to be tested. The EXPORT keyword may appear before the declaration of the class.

9. The start of the public member functions (and member data) are signaled by the "public" keyword. The keywords "protected" or "private," or the close of the class declaration "};" end the public section. C++ permits multiple public sections within a class declaration.

10. ICG 58 ignores enumerations, indicated by the keyword "enum."

11. Constructors. C++ permits more than one per class. C++ also permits constructors to be overloaded.

12. Copy Constructor. If provided, there is only one. A copy constructor has as its first required argument a reference (&) to an instance of this class (usually declared const). In a preferred embodiment, there is a second, optional argument, a memory manager defaulted to NULL. All classes must have one, but they can be private and unimplemented.

13. Destructor. The definition of most classes include a public destructor.

14. Assignment operator. All classes must have one, however, they can be declared private and not implemented.

15. Conversion operator. This operator, also known as cast, is provided in addition to the operators defined by C++. This operator returns an object of the type identified in the operator statement (e.g., dtULong). Also notice, this member function is const, which is not used in generating the signature of the inspection member function.

16. The addition assignment operator returns a reference to an object of this class type.

17. A static member function (sometimes called a class function) returning an erStatus object. A pointer to a production object is an output parameter. The static specifier is not included in the corresponding statement in the inspection class files 62a, 62b.

18. Public member data is ignored when generating the inspection class files 62a, 62b.

19. The start of the protected member functions (and member data) is signaled by the "protected" keyword. The keywords "public" or "private," or the close of the class declaration "};" ends the protected section. C++ permits multiple protected sections within a class declaration.

20. An overloaded index operator implemented as a const member function. Again, note that the const is ignored for the generated inspection function.

21. A pure virtual constant member function. This function signature is used to generate private virtual function declaration in the test class. This is done in order to satisfy the compiler that no attempt is being made to instantiate an abstract class. The pure (=0) specifi is not included in the test class declared in the inspection class header, but the const specifie must be preserved for this one case. No implementation is generated.

22. A virtual set member function with a default argument. The default argument is usec when generating the function signature of the inspection function. In this example, there exists a constant pointer to a constant object, with a default value of NULL.

23. The OPLDEBUG constant dump member function is defined for all objects. This member function is not duplicated in the generated inspection class.

24. Protected member data is ignored when generating the inspection class.

25. The start of the private member functions (and member data) are signaled by the private: keyword. The keywords public: or protected:, or the close of the class declaration } ends the private section. Multiple private sections are possible within a class declaration.

26. Private member functions and data are ignored when generating the inspection class files 62a, 62b.

27. Nonclass function appearing in the header for the production class. These do not appear often, and are usually a specialized function used by the production class.

28. Global nonclass operator appearing after the production class declaration. Similar ir use to nonclass functions.

29. Close of the include file guards.

Explanation of Annotations to Inspection Class Header File 62a Listing of Appendix B

1. Copyright notice comment copied from production class.

2. Include file guards are generated based on inspection class header filename (minus extension).

3. File name is i_utg.hh. The rules used to generate the filename are the same as those for the class. The name must be unique (in the output directory) so that existing inspectior classes are not overwritten. An optional parameter on the command line specifies the name the output filename.

4. The inspection base class declaration is included by the statement #include "idt-test.hh".

5. The production class header is included by the statement #include "utg.hh".

6. Inspection class comment. The !AUTHOR, !REVIEWER, and !REVIEW_DATE keywords and values are defaulted from a template (values from production class header are not used).

7. The inspection class comment keyword !LIBRARY value adds an "I_" prefix to the value from the production class comment (I_ER).

8. The inspection class comment keyword !NAME value adds a "i_" prefix to the name of the class (i_utg).

9. The inspection class generator 58 places a comment indicating that this file 62a is generated by a tool, to wit, the generator 58. The comment is entered in the !TEXT section of the class comment, and includes a date, the name of the tool, and the version of the tool.

10. The test class declaration publicly inherits from its production class to provide access to protected member functions. The test class name has "s_" prepended to the production class name. All member functions are declared in the "public:" section of the test class.

11. The test class declares the inspection class as a "friend." This allows the inspection class to invoke protected (and public) member functions on the production class.

12. Test class constructors and copy constructors are declared and implemented inline to replicate the public and protected interface on the production class. The private interface of the production class is not used.

13. Declaration to satisfy the compiler for a pure virtual member function declared in the production class. The declaration has lost its pure specifier (=0), but not its const function specifier. Also, the virtual specifier from the production class is also not needed (if it is easy to strip off, do it, otherwise the virtual specifier can stay). A comment is generated as an indication to the unit test developer that an inline implementation may be required (if linking causes this function to be unresolved, i.e., it is being called by some object).

14. An enum named FUNCS contains entries for each inspection function and a final entry named TOTAL_FUNCS. Each entry is composed of a prefix of f_ followed by the name of the inspection function. Where functions are overloaded (e.g., multiple constructors) it is necessary to append a number indicating which of the functions this enumeration applies. This enum is used for calculating base coverage analysis.

15. An integer array the size of TOTAL_FUNCS is used for based coverage analysis. Th name of this dtInt array is coverage.

16. The inspection class publicly inherits from the base test class idtTest.

17. The "public:" keyword is used to make all inspection class member functions available.

18. The inspection class itself requires a constructor and destructor. There is a difference between inspection class member functions and inspection member functions for the class under test. The member functions of the inspection class are needed to provide the behavior for the unit test as defined by idtTest and coding standards. The inspection member function are those functions needed to test the production class member functions.

19. The inspection test suite functions are more inspection class member functions. The implement the pure virtual functions defined in idtTest. These functions are invoked from th runTest member function of idtTest. The test functions are: t_LifeCycle, t_Operators, t_Semantics, t_SetQrys, and t_Persist. The qryCoverage member function is used to provid the coverage data for the base coverage analysis.

20. For each public or protected constructor provided by the production class there is a "newInstance" inspection member function. The newInstance member function takes the same arguments as the production class constructor and returns a pointer to an instance of th test class.

21. If the production class has a public or protected copy constructor, then a copyInstan member function is generated. This inspection member function takes as the first argument pointer to an instance of the test class, followed by the arguments of the production class' copy constructor (usually the memory manager). A pointer to an instance of the test class is returned. NOTE: the second parameter is optional and has a default value; this mirrors the production class copy constructor.

22. If there is a public or protected destructor declared on the production class, a deleteInstance inspection member function is generated. This inspection function takes one parameter, a pointer to an instance of the test class. There is no value returned (dtVoid).

23. Operator inspection member function naming. All operator functions must be mapp into a function name of the form "oper_<operator-name>." Above each declaration is a comment indicating its original name in the production class. In the annotations on Appen B, the term "ARM" refers to the Annotated C++ Reference Manual, authored by M. A. Ell and B. Stroustrup (Addison-Wesley, 1990).

24. Assignment operator inspection function. The first argument, as a general rule for all member functions, is a pointer to an instance of the test class, followed by the arguments from the production class assignment operator. The return type is modified to return a test class (s_utg&) instead of a production class (utg&). The suggested name for this operator is oper_equal.

25. The conversion operator must return an object of the same type as the conversion. Since a conversion operator itself has no arguments, the only argument becomes a pointer to an instance of the test class. The suggested naming for conversion operators is oper_<conversion-type>.

26. The addition assignment operator modifies the return type to a test class, and inserts a pointer to an instance of the test class, followed by the arguments defined by the production class. The suggested naming for this operator is oper_plusequal.

27. The static production member function has lost its static specifier in the inspection class. Also, production class parameter type has been replaced with the test class.

28. The index operator follows the same general rules as described for the assignment operator. The suggested name is oper_index.

29. The virtual setID member function contains a default argument, which is a constant pointer to a constant object. The virtual specifier from the production class is not needed on the inspection class (if it is easy to strip off then do it, otherwise the virtual specifier can stay). A pointer to an instance of the test class is inserted as the first argument, with the production class arguments following; including the default argument (=(const dtChar* const) NULL).

30. The start of the "private" section, and the end of the "public" section.

31. Declaration of the inspection class' copy constructor and assignment operator member functions. These are not implemented in the inspection class source file 62b.

32. Nonclass functions are declared outside of the inspection class. Their return type and arguments are preserved (a pointer to an idtTest object is inserted as the first argument, and production class types are modified to test class types). The name is changed so it does not conflict with the production header nonclass function name. A suggested naming scheme for nonclass functions is <inspection-class-name>_<nonclass-function-name>. For example, export becomes i_utg_export.

33. Global operators are declared outside of the inspection class. Return type and arguments are preserved (a pointer to an idtTest object is inserted as the first argument, and production class types are modified to test class types). The name is changed so it cannot conflict with the production global operator name. A suggested naming scheme is <inspection-class-name>_oper_<operator-name>. Therefore, the global production operator becomes i_utg_oper_plus in the inspection header.

34. The end of the include file guard.

Explanation of Annotations to Inspection Class
Source File 62b Listing of Appendix C

1. Copyright notice comment.

2. Name of this file. Follows naming rule identified in item 3 of the prior section.

3. Include the inspection class header.

4. Insert the TESTRUNdeclare macro text. Substitute in the name of the inspection cla (i_utg).

5. Insert the text for the comments, TESTRUN, and testname, substituting the name of the inspection class.

6. Class description comment, mostly a template with 3 substitutions.

7. Prefix the name of the library with an "I_".

8. Prefix the name of the class with an "i_".

9. Insert a string describing the date and version of the inspection class generator whicl generated this file.

10. Insert the class constructor with the coverage array initialization code, substituting tl name of the inspection class.

11. Destructor, leave empty, and substitute the name of the inspection class.

12. Lifecycle testing comment, insert as is.

13. Lifecycle testing initialization, insert as is, substituting the inspection class name.

14. Invoke newInstance using mm, substitute test class name. In further embodiments, a constructor/destructor test (TLIFE_START through TLIFE_END) can be generated for each existing production class constructor. Likewise, they may include argument types in where /*PARAMETERS,*/ currently appears.

15. Copy as is.

16. Invoke newInstance using mm, substitute test class name.

17. Invoke copyInstance using mm2, substitute test class name. In further embodiments, a constructor/destructor test (TLIFE_START through TLIFE_END) can be generated for each constructor/copy constructor combination. Likewise, they may include argument types in where /*PARAMETERS,*/ currently appears.

18. Copy as is.

19. Invoke newInstance using mm, substitute test class name.

20. Invoke newInstance using mm2, substitute test class name. In further embodiments, a constructor/destructor test (TLIFE_START through TLIFE_END) can be generated for each constructor/assignment operator combination. Likewise, they may include argument types in where /*PARAMETERS,*/ currently appears.

21. Copy as is.

22. Three invocations of newInstance using mm, mm2, and mm. Substitute test class name.

23. Copy as is.

24. Invoke newInstance using mm, substitute test class name.

25. Copy as is.

26. Operators testing comment, copy as is.

27. Operators testing body skeleton, substitute inspection class name for scope name (i_utg::t_Operators).

28. Comment and body for set and query tests. Same rules as requirement 27.

29. Semantics testing comments and body. Same rules as requirement 27.

30. Persistence testing comments and body. Same rules as requirement 27. In this case, the class is not persistent, so that no persistence testing is generated. Such testing is shown in the example below for the persistent class BasResc (see Appendix F and the accompanying text, *infra).*

31. Return coverage information. Copy as is, substituting inspection class name for scope

32. Comment for constructors (newInstance) for test class. Copy as is.

33. Generate function signature using production class constructor signature, substituting test class name (s_utg) for production class name (utg), using inspection class name for scope (i_utg). Applies to all inspection functions.

34. Generate coverage counter code (applies to all inspection functions). Use the scheme outlined in requirements 14 and 15 of section 3. The member function name text is used to substitute into coverage index text.

35. All inspection functions have an announceMethodStart call, substituting a unique name (requirement 34) based on the function as the argument ("newInstance1").

36. Invoke the corresponding constructor on the test class, substituting the test class name for the production class name.

37. All inspection functions have an announceMethodEnd call, following the rules outlined in requirement 35.

38. All newInstance inspection functions return a pointer to an instance of the test class.

39. Invoke the test class copy constructor in copyInstance, dereference aInstance. Substitute test class name where appropriate.

40. The deleteInstance member function must generate a delete statement.

41. Generate function signatures for other inspection functions using same rules as outlined in requirement 33.

42. Invoke the corresponding member function on the test class.

43. If the inspection function returns a value, generate the return statement using either the aInstance pointer or the testVal variable.

44. Another version of requirement 42, this time the corresponding test class member function invoked returns an object. Declare testVal of the correct type.

45. Another version of requirement 43, this time a testVal is being returned.

46. Nonclass functions have a pointer to an idtTest passed in as the first argument. This pointer is used for invoking the announceXxxx member functions. This is done because nonclass functions are not member functions on an inspection object (an inspection object is a idtTest object).

Generation of the inspection class header and source files 62a, 62b by the ICG 58 may be further understood by reference to the Appendix D, providing a listing of sample production class header 60 named "basresc.hh"; Appendix E, providing a listing of a sample inspection class header file 62a named "i_basres.hh" that is generated by the ICG 58 from the file basresc.hh; and, Appendix F, providing a listing of a sample inspection class code file 62b named i_basres.cc that is also generated by the ICG 58 from the file basresc.hh.

The code generation operations revealed Appendices D - F include support for basic persistence testing of a persistable (or persistent) object is the body using the "t_persist()" function and two additional support member functions for setting up valid user keys and persistent objects.

The items listed below refer to the annotations made on the listings in Appendices D - F. In the description of each item, reference is made to the three classes involved in the unit test:

1. The source production class 30, which is a concrete persistent class.

2. The test class 40. This is a generated typedef which provides access to the public member functions of the production persistent class.

3. The inspection class 34. This class performs tests on instances of the test class 40.

As above, where dictated by context, references in the explanatory items below to the "production class," the "inspection class" and the "test class" shall be interpreted references to the corresponding files 60, 62a, 62b.

Explanation of Annotations to Production Class
File 60 Listing of Appendix D

1. User Key class name. A user key is a class that contains text which is the name of a persistent object. This can be obtained from scanning the second parameter type of the openForChange(), openForReview(), and saveAs() member functions. In an alternate embodiment, a hint as to the icg is obtained from the production class developer via a !USERKEY keyword.

Explanation of Annotations to Inspection Class
Header File 62a Listing of Appendix E

1. User key setup member function signature. The setupUKs() member function takes four arguments: three user keys and a memory manager pointer. Copied as is except where noted in 2.

2. User key class name is used for each of the three user key arguments. This is the onl item that needs to be substituted in this function signature.

3. persistent object setup member function signature. Takes two arguments: test class pointer and a memory manager pointer. Copied as is except where noted in 4.

4. Test class name is the only item that must be substituted.

Explanation of Annotations to Inspection Class
Source File 62b Listing of Appendix F

1. Define the test class name. This must be substituted using the generated test class name.

2. Define the object reference template parameter. The value that must be substituted i the name of the production class within the angle brackets (oObjRef<ProdClass>).

3. Copy other definitions as is, only STDGIT_PRODUCTION should not be commented.

4. Declare three User Key objects. Substitute the class name of the user key.

5. Declare other test objects. This includes three persistent instances, a security contex and a commented kind reference.

6. Declare a kind reference with its template parameters. Declares a variable named "k of the appropriate kind reference. A kind reference is a templated class, and this item is calling out the production class name that must be substituted in the first template paramet oObjRef (which is itself a templated class). Similar to item 2 in this section.

7. The second kind reference template parameter must be substituted with the user ke class name.

8. The rest of the t_persist member function body is copied as is. Note the inclusion the templated persistence test text.

9. The qryCoverage member function, which is currently being generated.

10. The user key setup member function body, setupUKs(). This is copied as is, except where noted in 11.

11. User key arguments for setupUKs() requires substitution of the user key class name.

12. The persistent object setup member function body, setupObj(). This is copied as is, except where noted in 13.

13. Test class pointer argument for setupObj() requires substitution of the test class name.

UNIT TEST HARNESS

With continued reference to **FIGURE 9**, the inspection class source file 62b generated by ICG 58 is compiled, linked and loaded (see element 64), along with production class header and source files 60, 68, as well as library files containing the test monitor class definitions 70 and the test kernel class definitions 72. Compiling, linking and loading is performed in a conventional manner, e.g., using a C++ compiler sold by Microsoft Corporation. The resulting file 66 containing executable code for the unit test harness is executed on digital data processor 12 to reconfigure that processor as the unit test harness.

FIGURE 10 depicts operation of the unit test harness 74. Particularly, testmain() routine 76 creates an inspection object 78 as an instantiation of the inspection class 34 defined in files 62a and 62b. Testmain() also invokes testrun(). The testrun() method 79 of the inspection object invokes test suite members 50a - 50e of the inspection object, each of which (i) creates a test object 80 instantiating the test class 40 (defined in file 62b), (ii) invokes a members of the test class, e.g., 42a, via corresponding inspection members, e.g., 36a, of the inspection class, and (iii) utilizes reporting members 82 of the inspection object 78 to report results of the invocation of the test class members. Reports can be generated to printer 25, to monitor 24, to a file on disk drive 22, or in other media as known in the art.

As illustrated, the test harness 74 accepts arguments, e.g., from the user, for application to the test object upon or upon invocation of the selected member methods, e.g., 42a. Such arguments can, alternatively, be generated automatically be the test harness 74.

Reports generated by members 82 of the inspection object include those resulting from comparing the results of invocation of the test object member, e.g., 42a, with expected result values. In the case of disagreement, report members 82 cause an error message to be displayed, e.g., to the printer, monitor or a disk file. Report members 82 also permit the data members of the test object 80 to be "dumped" following invocation. The report members 82

store data reflecting errors that occur during invocation of the test object members, e.g., 42a, and coverage of the testing (i.e., which test object members are tested).

UNIT TEST CLASSES

idtkrnl

Report member 82 includes macros inherited from class *idtKrnl* 48 (Figure 5) providing unit test output. Significant among these are two that provide for error checking and a third that gives the developer the ability to output their own status message:

checkExpr(expression, testname) - Announces an error if the expression evaluates to FALSE, returning the value of the expression.

checkMemMgr(memoryMgr, testname) - Announces an error if the Memory Manager detects a memory leak. Returns TRUE if no leak was detected, FALSE when there is a leak.

announceStatus(message, testname) - Announces the message in standard unit test format.

These macros automatically report the standard output information such as current fil and line. The report members 82 also include specific announcement members inherited fror class *idtKrnl* 48, including *announceObject()*, which dumps the contents of the object to the output stream.

idtKrnl 48 also provides other announcement methods:

announceMethodStart(methodname) - Reports the start of the named method

announceMethodEnd(methodname) - Reports the end of the named method

announceParameter(parameterObj) - Dumps the contents of the object, under the label "Parameter"

announceRetVal(returnObj) - Dumps the contents of the object, under the label "Return Value"

idtTest

The superclass *idtTest* 46 also provides to inspection object 78 mechanisms for executing unit tests and monitoring test coverage. In this regard, the inspection object inheri from *idtTest* 46 the following macros to standardize the start and end of the test suite metho 50a - 50e (also referred to as t_* methods):

T_INIT(methodname) - Initialize a t_* method, setting up a test memory manager, mm

T_CLEANUP(methodname) - Clean up a t_* method, checking the memory manager for leaks.

TEST_START(testname) - Start a test scenario

TEST_END - End a test scenario; announces whether the scenario passed or failed.

TEST(testname, expression) - A combination of TEST_START() and TEST_END for tests which consist of a single expression to check.

checkExcept(expression, id, testname) - Verify that evaluating the expression throws an exception with the id indicated. *CheckExcept* is implemented in systems that do not otherwise support exception handling by the use of the *setjump* and *longjump* ANSI functions. Prior to evaluating the expression, *setjump* is performed to save the current stack frame. If an exception is thrown and it is required to restore the stackframe, *longjump* is performed.

In addition to the foregoing, the inspection object inherits from *idtTest* the testrun() method and other code for running the t_* methods and the code for analyzing the results of the unit tests.

DEVELOPING UNIT TESTS

The sections that follow provide still further discussion of the operation of the inspection class generator 58 and unit test harness 74. Also discussed is designer modification of the inspection class header and source files 62a, 62b for completing test suites for the test harness.

Although inspection class generator 58 creates most of the unit test code 66 from the production class header file 60, the scope of testing may necessitate that the developer complete implementation of the test and inspection classes, as well as the test scenarios themselves. As noted above, in order to manage the potentially large number of test scenarios, the methods are divided into the following logical groups for testing:
LifeCycle - creation and destruction of the object (tests for memory leaks)
Set & Query - Accessors and transformers of the object
Operators - Operator methods
Persist - methods involved in object persistence
Semantics - methods unique to the class.

Each of these groups are implemented in the inspection class as a method 50a - 50e whose name is the class prefixed by "t_", such as t_LifeCycle. In order to preserve the isolation of the methods, each test case must be able to be executed on its own, without depending on other test cases running before or after it, although test cases may share test data.

Testing Object Creation and Deletion — t_LifeCycle

The purpose of t_LifeCycle is to test the apparent operation of the constructors and destructor of the object, including the copy constructor. In addition, because it interacts closely with the constructors, the basic assignment operator (=) is tested in t_LifeCycle. The major focus of t_LifeCycle is to catch memory leaks.

Because memory leaks can not be cleared from a memory manager, it is necessary to create a new manager for each test case. In addition, since the memory manager can only check for memory leaks if there are no outstanding blocks of memory allocated, all objects constructed using this memory manager must be deleted before the end of the test. A convenient way to do this is to enclose the test case within a block ({ }), that way, local variables are deleted at the end of the block as they go out of scope.

Special TLIFE_* macros in *idtTest* 46 manage the creation of memory managers and the block. These macros are:

TLIFE_INIT - Initialize t_LifeCycle()

TLIFE_START(testname) - Starts a LifeCycle test case by creating two memory managers (mm and mm2) and starting a block

TLIFE_END - Ends a LifeCycle test case by closing the block and checking the memory managers for leaks (before destroying them).

Within the scope of the test, two memory managers are available, in the variables mm and mm2. Constructor tests only need one memory manager, but the copy constructor and assignment constructor should be tested with two memory managers to ensure that the data completely copied from one object to another. This provides added memory integrity in a threaded environment.

Constructor/Destructor Testing

It is necessary to have at least one test case for each constructor. The strategy for a constructor test is as follows:

1. construct a test object
2. announce the test object
3. delete the test object

Note that the destructor is tested along with the constructor in these cases. The basic test is illustrated in Appendix G.

Also note the use of the test class, s_erMsg, in this example. The test class should be used to represent the production class in all unit test code.

Another consideration in constructor operation is their behavior under error conditions. Usually this means testing constructors when there is insufficient memory for the new object.

In one embodiment, out-of-memory exceptions are thrown by the memory manager, and caught at the application level, so object constructors are not involved. However, constructors may require parameter testing.

Parameter testing should be performed when a parameter's value has some restrictions which are enforced by the constructor. In this case, use equivalence partitioning and boundary value analysis to determine the necessary test cases. For example, if an object had a member which was an unsigned integer, but the only legal values were in the range 1 - 100. In this case, test cases need to include calling the constructor with the values 0 (invalid), 1, 50, 100, 101 (invalid) and 500 (invalid) might be used.

If the parameters are pointers, test cases should be written to check behavior when the parameter is NULL. A test case to check the default constructor needs to be written as well.

Copy Constructor Testing

Testing the copy constructor follows the form of tests for the regular constructors, with slight modifications. In particular, the constructed object should be in a different memory area than the copied object to ensure that the copy is completely independent of the original. Referring to Appendix H, the scenario is as follows:

1. construct an object (obj1)with memory manager mm
2. construct the test object, by copying (obj2)with memory manager mm2
3. check the equality of the objects (if an equality operator exists) (obj1==obj2)
4. delete the original object (obj1)
5. announce the test object (obj2)
6. delete the test object (obj2)

Note that deleting the original object catches errors involved with incomplete copying. TLIFE_END check both memory managers for leaks. The copy constructor needs to have a test case for each constructor.

Assignment Operator Testing

The assignment operator is tested similarly to the copy constructor, although the basi scenario is slightly different:

1. construct an object (obj1) with memory manager mm
2. construct the test object (obj2) with memory manager mm2
3. assign the first object to the second object (obj2 = obj1)
4. check the equality of the objects (if an equality operator exists) (obj1==obj2)
5. delete the original object (obj1)
6. announce the test object (obj2)
7. delete the test object (obj2)

The assignment operator needs to have a test scenario for each constructor, performing the assignment across memory areas. Additionally, the assignment operator need two more test scenarios:

Chains of assignment (obj3 = obj2 = obj1, checking that obj3 == obj1)

Assignment to self: (obj1 = obj1, checking that the obj1 still contains its value)

Note that this tests assignment of two like objects; assignment of an object of anothe type to a production class object is tested in t_operators.

Example LifeCycle Test Design

Consider a class with three constructors: a default constructor, a constructor with 3 dtLong parameters, and a constructor with 3 dtLong and a bcStringi18n parameters, and a copy constructor and an assignment operator. It would have twelve test scenarios, as follows:

1. Default constructor test
2. Constructor 2 test - values (0, 0, 0)
3. Constructor 3 test 1 - values (0, 0, 0, "")
4. Constructor 3 test 2 - values (1, 2, 3, "abc")
5. Copy constructor test 1 - copying object from default constructor
6. Copy constructor test 2 - copying object from constructor 2
7. Copy constructor test 3 - copying object from constructor 3
8. Assignment operator test 1 - assigning object from default constructor
9. Assignment operator test 2 - assigning object from constructor 2
10. Assignment operator test 3 - assigning object from constructor 3
11. Assignment operator test 4 - chain of assignment
12. Assignment operator test 5 - assignment to self

Testing Set and Query Methods — t_SetQrys

The purpose of t_SetQrys is to test the operation of the accessors and transformers of the class; this includes the Query, Set and Validate methods.

Query Methods Testing

The query methods should be tested first, because they can then be used to test the Set methods. There are two approaches to testing the Query methods. If the method returns an unaltered value, the approach is to create an instance of the class with a known value in the member(s) being queried, and then check that the query returns these values, as illustrated in Appendix I.

Note the use of the TEST() macro; the TEST() macro combines TEST_START(), checkExpr() and TEST_END macros, allowing a test which consists of a single expression to be tested in a single line. Since the example test case consists of the expression, "fdMsgTest3 == qryMsgID(msg1)", the test can be implemented as a call to this macro.

The number of test cases that are needed to validate a query method depends on the complexity of the query and boundary conditions. In some queries, the member being queried may not be present; remember to test both the positive (member present) and the negative (member absent) cases, as illustrated in Appendix J. Null and 0 are the most common boundary values; always consider whether test cases for these need to be written.

Another consideration occurs when the value being queried undergoes some transformation either at creation or during the query. In this case, additional test cases would need to check the special cases, assumptions, or boundaries that the calculation imposed on the data. For example, a date class which stored its value in terms of month, day, year, but included a method qryDayOfYear() which calculates the number of days since the start of the year. This method adds to the class of valid test data the conditions of leap and non-leap years (and skipped leap years at century boundaries), and the special case of the year 1752; test cases need include February 29 and December 31, for both a leap and non-leap years, dates in the skipped leap years, such as 1900, and dates in 1752 (depending on locale).

There are no restrictions in setting up test data which satisfies several query unit tests at once, as shown in Appendix J.

Set Methods

The approach to testing the set methods is to apply the method and check the results with the appropriate query method(s). Test cases need to be written to test all of the boundary conditions in the same manner as for query methods described above.

Testing Operators — t_Operators

The set of all possible operators is large; all of the operators are considered here, for completeness. However it is expected that for any particular class only a few operators will be defined.

The general approach for operator testing is to use equivalence partitioning and boundary values to establish a list of test inputs and compare the result of the operation with constant value. As with query and set methods, the comparison operators should be checked first, so that they can be used to test the other operators.

In the inspection class, the instrumented versions of the operators become procedural methods, so they are renamed, for example, "operator+" becomes "oper_plus".

Note that negative testing of operators will likely result in the production code raising exceptions.

Conversion Operators

The conversion operators are casts which convert the production class into another class. In many cases there is a natural mapping between the class and the target class; even so, make sure to check the 0/Null boundary. Where a natural mapping does not exist, check the code for special cases and equivalence classes.

Conversion operators are renamed in the inspection class to "oper_type", where type is the result of the cast. So in our example, the cast to dtLong* is renamed oper_dtLong in the inspection class. The original, in erMsg is:

```
//////////////////////////////////////////////
// !METHOD_DECL_S
// !NAME erMsg_operator_dtULong()
// !SATISIFIES
// 1)
// !TEXT
// erMsg conversion operator.  Causes the
// value of the error message to be returned.
// Shorthand for qryMsgID().
//////////////////////////////////////////////
```

```
operator dtULong() const;

This is tested in i_erMsg as follows:
//
// Test 1: Cast to dtULong
//
const dtULong test1value = 1;

s_erMsg* msg1 = newInstance( test1value, ERMSGTEST_NAME, ERMSGTEST_5,
__FILE__, __LINE__, mm );

        // Do cast and check the value
TEST( "Cast to dtULong", test1value == oper_dtULong( msg1 ) );

deleteInstance( msg1 );
```

Testing Object Persistence — t_Persist

The purpose of t_Persist is to test the object persistence for the class: this includes the Persist and Fetch methods.

Testing Unique Methods — t_Semantics

The purpose of t_Semantics is to test those methods unique to the class.

Test Execution Code

The inspection class generator 58 creates the code which allows the single unit test harness 74 to run the tests. The harness 74 collects arguments from the user to run the test. Then, the test harness 74 calls two exported routines: *testname()* and *testmain()*, to figure out what is being tested and to execute the tests. *testname()* returns a dtChar * which names the inspection class. *testmain()* serves as the top level of test execution, calling testrun() which, in turn, calls inspection class methods which test the production class. In the inspection class, two macros are used, *TESTRUNdeclare* and *TESTRUN* (supplied in idtTest 46), so that *testmain* is not coded directly. Implementation of these routines is illustrated in Appendix K in which the class "erMsgi" refers to the class "i_erMsg."

Once the inspection class code has been written, it is built into a dynamic link library, or "DLL." Each inspection class must have its own DLL, so that the *testname()* and *testrun()* routines do not clash.

RUNNING UNIT TESTS

Using the Unit Test Harness

In a preferred embodiment, when the unit test harness 74 is started up, it brings up th window shown in FIGURE 11. The user fills in the first text field with the name of the DLI built with the inspection class (omitting the .dll extension), and the second text field with the name of the file to which report output is to be directed. The user also selects the desired verbosity level for each test suite member method and hits the "Run" button.

The unit test harness 74 then loads the inspection class DLL, and executes the *testname()* and *testrun()* routines to perform the unit tests. When the tests are complete, the "Error count" field is filled with the number of errors detected during the run. Pressing the View Results buttons will bring up a multi-document notepad with the various output files.

OUTPUT OF UNIT TESTS

Types of Unit Test Output

Aside from counting the number of errors detected, the unit test code 74 outputs messages in three categories: error messages, status messages and debugging information. The first class of messages demonstrates whether the production code has correct behavior: they are generated by checkExpr(), checkExcept(), checkMemMgr(), TEST(), and TLIFE_END macros when an expression fails to evaluate to TRUE or when a memory leak is detected. The second type of messages allows the execution of the unit tests to be traced and they are generated by the TEST_START(), TEST_END, TLIFE_START(), TLIFE_EN macros and by code generated by the inspection class generator. The third class enables debugging; these messages are generated by the announceObject and code added by the developer.

A user may need some or all of these messages, depending on their goal in executing the unit tests. If they only wish to determine if the production code is operational, for example, they may just want the error messages. In order to give the user some control over the classes of messages they receive, each message has an associated "verbosity level." If th current verbosity level is equal to or higher than the level of the message, that message is pu into the output stream. This is the effect of the verbosity widget in the unit test harness; by setting the verbosity higher or lower, the user can control what message they get. The verbosity level of the unit test messages are listed below.

Verbosity	Class	Sample Messages
SKIP	Skip particular test suite	SKIPPING: t_LifeCycle, verbosity set to SKIP
OFF	None	
LOW	Error messages	ERROR: in test FOO, file FILE, line LINE, bad expression: EXPR
MED	Status messages	Starting test FOO PASS: FOO
HIGH	Object dumps, other debug messages	Current value of mDate: FOO

If the user sets the verbosity level to OFF, the unit tests will not output any messages, but if they set the verbosity level to MED, they will get the error and status messages in the output log file. Appendices L, M and N depict of test runs at the various verbosity levels. SKIP allows execution of a test suite to be skipped, e.g., for purposes of initial test development and debugging.

In addition, the inspection class appends a simple analysis of the run at the end of a unit test execution. testmain() always returns the number of errors detected, so the test harness can display this information even with the verbosity level set to OFF. At LOW and MED verbosity levels, the unit test code 74 reports on the number of tests run and the number of errors found. At HIGH, coverage information is output; this allows the user to determine whether all of the production methods were tested.

Unit Test Output Files

Output by harness 74 and, particularly, by reporting members 82 is placed into five files:

Log file: <file>.out (<file> represents the destination file field in the unit test harness)
Error file: <file>.err
Memory Log file: memory.log
Exception Log file: except.log
Database Error file: sqlerror.log

The log file is the main file for unit test output. All of the output generated by the unit test code is placed into this file, giving the user a place where they can read the messages in context. As described above, the contents of the log file is controlled by setting the verbosity level.

The error file separates the error messages from the rest of the output for easier management. It is not affected by the verbosity level; it always contains all of the error

messages generated by the unit test code. The memory log file includes information output b the memory management code, which can be useful in tracking down memory leaks. The exception log includes information on the exceptions which were thrown by the unit test code.

GENERATION TECHNIQUES

In an alternative embodiment, as inspection class 34 as described comprises an "inspection standard" class and an "inspection custom" class, the latter (i.e., the custom clas deriving from the former (i.e., the standard class). The inspection standard class includes al of inspection methods 50F (Figure 6), while the inspection custom class includes the test suites 50A - 50E. Those skilled in the art will appreciate that this facilitates maintenance of the system and, particularly, facilitates regeneration of the inspection standard class as necessitated when the API of the production class changes.

SUMMARY

Described above are improved methods and apparatus for testing object-oriented programming constructs. It will be appreciated that the illustrated embodiment is described by way of example and that other embodiments incorporating modifications may well fall within the scope of the invention. Thus, for example, it will be appreciated that class name function names and variable names may differ from those used herein. In view of the foregoing, what we claim is:

```
////////////////////////////////////////////////////////////////////////
//
// FILE_NAME: utg.hh
//
// Copyright 1993 by Marcam Corp., Newton, MA USA
//
// This unpublished copyrighted work contains
// TRADE SECRET information of Marcam Corporation.
//
// Use, transfer, disclosure, or copying without
// its expressed written permission is strictly
// forbidden.
//
//
////////////////////////////////////////////////////////////////////////

#ifndef utgHH
#define utgHH

#include "dtcore.hh"          // For dtCore

////////////////////////////////////////////////////////////////////////
// !CLASS_DECL_S
// !LIBRARY EP
// !NAME utg
// !TEXT
// Test case for the unit test generator.  This class header attempts
// to test a number of different items that need to be transformed
// when creating the inspection class header.  This includes the
// public and protected member functions, operators, static and nonstatic
// member functions, and nonclass functions appearing in a header file.
// Also, various styles of using spaces in a member function delcaration
// are tested.
//
// !AUTHOR      John Dalton          (dalton@opl.com)
// !REVIEWER    <Reviewer's name> <(Reviewer's E-mail address)>
// !REVIEW_DATE <date>
////////////////////////////////////////////////////////////////////////

class EXPORT utg : public dtCore
{

public:
    // Public enumeration should not end up in inspection class.
    enum stringSize
    {
        STRING_SIZE = 10
    };


    ////////////////////////////////////////////////
    // !METHOD_DECL_S
    // !NAME utg_def_ctor
    // !TEXT
    // The default constructor.  Most classes are
    // required to have one.
    ////////////////////////////////////////////////

    utg( dtMemoryMgr* aMemMgr );

    ////////////////////////////////////////////////
    // !METHOD_DECL_S
    // !NAME utg_ctor1
```

UTG.HH

```
// !TEXT
// This constructor takes 3 arguments, all of which are required.
//////////////////////////////////////////////////

utg( dtULong              aArg1,
     const dtChar* const aArg2,
     dtMemoryMgr*         aMemMgr );

//////////////////////////////////////////////////
// !METHOD_DECL_S
// !NAME utg_ctor2
// !TEXT
// This constructor takes 4 arguments, all are required.
//////////////////////////////////////////////////

utg( dtULong              aArg1,
     const dtChar* const aArg2,
     dtInt                aArg3,
     dtMemoryMgr*         aMemMgr );

//////////////////////////////////////////////////
// !METHOD_DECL_S
// !NAME utg_copy_ctor
// !TEXT
// utg copy constructor.
//////////////////////////////////////////////////

utg(const utg&   aUtg,
    dtMemoryMgr* aMemMgr=(dtMemoryMgr*)NULL);

//////////////////////////////////////////////////
// !METHOD_DECL_S
// !NAME utg_dtor
// !TEXT
// utg destructor.  Does nothing special.
//////////////////////////////////////////////////

~utg();

//////////////////////////////////////////////////
// !METHOD_DECL_S
// !NAME utg_operator=
// !TEXT
// utg assignment operator.
//////////////////////////////////////////////////

utg& operator= ( const utg& aRhs );

//////////////////////////////////////////////////
// !METHOD_DECL_S
// !NAME utg_operator_dtULong()
// !TEXT
// utg conversion operator.
//////////////////////////////////////////////////

operator dtULong() const;

//////////////////////////////////////////////////
// !METHOD_DECL_S
// !NAME utg_operator+=()
// !TEXT
// An overloaded Addition Assignment operator.
//////////////////////////////////////////////////
```

11

12

13

14

15

16

8/17/93 7:04PM Page 2

UTG.HH

```
   utg& operator += ( dtInt aRhs);

   ////////////////////////////////////////////////
   // !METHOD_DECL_S
   // !NAME utg_openForRead
   // !TEXT
   // Static method generates a utg.
   // Static specifier is dropped on inspection
   // member function.
   ////////////////////////////////////////////////

   static erStatus openForRead( utg*& aUtg, dtMemoryMgr* aMM);

   // Public member data.  Should not appear in generated inspection class.
   dtUInt       publicID;
   dtChar       publicIDstring[STRING_SIZE];
protected:

   ////////////////////////////////////////////////
   // !METHOD_DECL_S
   // !NAME utg_operator[]()
   // !TEXT
   // An overloaded index operator.
   ////////////////////////////////////////////////

   dtChar operator []( dtInt aIndex ) const;

   ////////////////////////////////////////////////
   // !METHOD_DECL_S
   // !NAME utg_rtnID
   // !TEXT
   // Returns ID. A pure virtual member function.
   // makes this an abstract class.  These don't
   // get tested by the unit test, but must be
   // declared (as private) in the inspection class
   // to satisfy compiler (instantiating an abstract
   // class).  No source is generated for this, it
   // can be declared and implemented.
   ////////////////////////////////////////////////

   virtual dtULong rtnID() const = 0;

   ////////////////////////////////////////////////
   // !METHOD_DECL_S
   // !NAME utg_setID
   // !TEXT
   // Set the ID, has an optional parameter.
   // Virtual function declaration is not used
   // on generated inspection member function.
   ////////////////////////////////////////////////

   virtual dtBoolean setID( const dtULong aID,
                            const dtChar* const aName = (const dtChar* const) NULL );

#ifdef    OPLDEBUG
   ////////////////////////////////////////////////
   // !METHOD_DECL_S
   // !NAME utg_dump
   // !TEXT
   // This is used by the announceXXX() methods, so
   // there is no need to include this in the
   // generated inspection class.
   ////////////////////////////////////////////////

   dtVoid dump(dtOStream& aOutStream, dtUInt aIndentLevel = 0) const;
```

```
#endif // OPLDEBUG

   // Protected member data.  Should not appear in generated inspection class.
   dtUInt       protectedID;
   dtChar       protectedIDstring[STRING_SIZE];

private:
   // Of course, nothing private will appear in the inspection class.
   dtULong          mMsgID;

   ////////////////////////////////////////////////
   // !METHOD_DECL_S
   // !NAME utg_privateMemberFunction
   // !TEXT
   // Private member functions do not appear in the
   // generated inspection class.
   ////////////////////////////////////////////////

   const utg* copy( const utg* const aUtg ) const;
};

// Nonclass function appearing in the header.
// These appear in the generated header prepended
// with the name of the inspection class.

dtBoolean export( const utg& aUtg );

// Global operator (also a nonclass function)

utg& operator+( const utg& aLhs, const utg& aRhs );

#endif // utgHH
```

24 25 26 27 28 29

41

I_UTG.HH

```
////////////////////////////////////////////////////////////////////////
//
// FILE_NAME: i_utg.hh
//
// Copyright 1993 by Marcam Corp., Newton, MA USA
//
// This unpublished copyrighted work contains
// TRADE SECRET information of Marcam Corporation.
//
// Use, transfer, disclosure, or copying without
// its expressed written permission is strictly
// forbidden.
//
//
////////////////////////////////////////////////////////////////////////

#ifndef i_utgHH
#define i_utgHH

#include "idttest.hh"          // For idtTest
#include "utg.hh"              // For utg

////////////////////////////////////////////////////////////////////////
// !CLASS_DECL_S
// !LIBRARY I_ER
// !NAME i_utg
// !TEXT
// Generated 7/12/93 by wutg version 1.0
//
// !AUTHOR      <Author's name>   <(Author's E-mail address)>
// !REVIEWER    <Reviewer's name> <(Reviewer's E-mail address)>
// !REVIEW_DATE <date>
////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////
// Class under test.  This class is derived from the production object.
// This provides access to the public and protected member functions
// of the production class, without modification to the production
// class.  Also, pure virtual functions declared in the production class
// are provided an implementation here.  This allows production abstract
// base classes to be instantiated and tested.
////////////////////////////////////////////////////////////////////////

class s_utg : public utg
{
public:
    friend class i_utg;

    ////////////////////////////////////////////////
    // Inline constructors, and copy constructor
    // for replicating the production class interface.
    ////////////////////////////////////////////////

    s_utg( dtMemoryMgr* aMemMgr ) : utg( aMemMgr )()

    s_utg( dtULong aArg1,
           const dtChar* const aArg2,
           dtMemoryMgr* aMemMgr ) : utg( aArg1, aArg2, aMemMgr )()

    s_utg( dtULong aArg1,
           const dtChar* const aArg2,
           dtInt aArg3,
           dtMemoryMgr* aMemMgr ) : utg( aArg1, aArg2, aArg3, aMemMgr )()

    s_utg( const s_utg& aS_utg,
```

8/18/93 12:52AM Page 1

42

I_UTG.HH

```
        dtMemoryMgr* aMemMgr=(dtMemoryMgr*)NULL ) : utg( aS_utg, aMemMgr ){}

    ////////////////////////////////////////////////
    // Provide an implementation for pure virtual
    // functions declared by the production class.
    ////////////////////////////////////////////////

    virtual dtULong rtnID() const;
    //!FIX ( If this is called you need to add code here )

};


enum FUNCS
{
    f_newInstance1,
    f_newInstance2,
    f_newInstance3,
    f_copyInstance,
    f_deleteInstance,
    f_oper_equal,
    f_oper_dtULong,
    f_oper_plusequal,
    f_oper_index,
    f_setID,
    f_openForRead,
    f_i_utg_export,
    f_i_utg_oper_plus,
    TOTAL_FUNCS
};

dtInt coverage[TOTAL_FUNCS];


class EXPORT i_utg : public idtTest
{

public:

    ////////////////////////////////////////////////
    // Inspection class member functions:
    ////////////////////////////////////////////////

    i_utg( dtStreamExecutive* aStreamExec,
           dtMemoryExecutive& aMemExec,
           idtTestArgs&       aTestArgs,
           dtMemoryMgr*       aTestMM );

    ~i_utg();


    ////////////////////////////////////////////////
    // Inspection test suite.
    ////////////////////////////////////////////////

    virtual dtVoid t_LifeCycle();
    virtual dtVoid t_Operators();
    virtual dtVoid t_Semantics();
    virtual dtVoid t_SetQrys();
    virtual dtVoid t_Persist();
    virtual dtVoid qryCoverage( dtInt*& aCovAry, dtInt& aNumFuncs ) const;


    ////////////////////////////////////////////////
    // Constructors for class under inspection
    ////////////////////////////////////////////////
```

8/18/93 12:52AM Page 2

43

I_UTG.HH

```
    s_utg* newInstance( dtMemoryMgr* aMemMgr );

    s_utg* newInstance( dtULong             aArg1,
            const dtChar* const aArg2,
            dtMemoryMgr*        aMemMgr );

    s_utg* newInstance( dtULong             aArg1,
            const dtChar* const aArg2,
            dtInt               aArg3,
            dtMemoryMgr*        aMemMgr );

    ////////////////////////////////////////////////
    // Copy constructor for class under inspection.
    ////////////////////////////////////////////////

    s_utg* copyInstance( s_utg* aInstance,
          dtMemoryMgr* aMemMgr=(dtMemoryMgr*)NULL);

    ////////////////////////////////////////////////
    // Destructor for class under inspection.
    ////////////////////////////////////////////////

    dtVoid deleteInstance( s_utg* aInstance );

    ////////////////////////////////////////////////
    // Other member functions.
    ////////////////////////////////////////////////

    s_utg& oper_equal( s_utg* aInstance,
            const s_utg& aRhs );

    dtULong oper_dtULong( s_utg* aInstance );

    s_utg& oper_plusequal( s_utg* aInstance,
            dtInt aRhs );

    erStatus openForRead( s_utg*& aUtg, dtMemoryMgr* aMM);

    dtChar oper_index( s_utg* aInstance,
            dtInt aIndex );

    dtBoolean setID( s_utg* aInstance,
            const dtULong aID,
            const dtChar* const aName = (const dtChar* const) NULL );

private:

    ////////////////////////////////////////////////
    // Inspection class member functions, unimplemented
    // copy constructor and assignment operator.
    ////////////////////////////////////////////////

    i_utg( const i_utg& aI_utg );

    i_utg& operator=( const i_utg& aI_utg );
};

////////////////////////////////////////////////
// Nonclass functions to be tested.
////////////////////////////////////////////////

dtBoolean i_utg_export( idtTest* aTestObj, const s_utg& aUtg );

s_utg& i_utg_oper_plus( idtTest* aTestObj, const s_utg& aLhs, const s_utg& aRhs );

#endif // i_utgHH
```

20 21 22 23 24 25 26 27 28 29 30 31 32 33 34

I_UTG.HH AMENDMENT

```
    s_utg* copyInstance( s_utg* aInstance,
          dtMemoryMgr* aMemMgr=(dtMemoryMgr*)NULL);

    ////////////////////////////////////////////////
    // Destructor for class under inspection.
    ////////////////////////////////////////////////

    dtVoid deleteInstance( s_utg* aInstance );

    ////////////////////////////////////////////////
    // Other member functions.
    ////////////////////////////////////////////////

    // "oper_equal" tests "operator="
    s_utg& oper_equal( s_utg* aInstance,
          const s_utg& aRhs );

    // "oper_dtULong" tests "operator dtULong()"
    dtULong oper_dtULong( s_utg* aInstance );

    // "oper_plusequal" tests "operator+="
    s_utg& oper_plusequal( s_utg* aInstance,
          dtInt aRhs );

    erStatus openForRead( s_utg*& aUtg, dtMemoryMgr* aMM);

    // "oper_index" tests "operator[]"
    dtChar oper_index( s_utg* aInstance,
          dtInt aIndex );

    dtBoolean setID( s_utg* aInstance,
          const dtULong aID,
          const dtChar* const aName = (const dtChar* const) NULL );

private:

    ////////////////////////////////////////////////
    // Inspection class member functions, unimplemented
    // copy constructor and assignment operator.
    ////////////////////////////////////////////////

    i_utg( const i_utg& aI_utg );

    i_utg& operator=( const i_utg& aI_utg );
};

////////////////////////////////////////////////
// Nonclass functions to be tested.
////////////////////////////////////////////////

dtBoolean i_utg_export( idtTest* aTestObj, const s_utg& aUtg );

_utg& i_utg_oper_plus( idtTest* aTestObj, const s_utg& aLhs, const s_utg& aRhs );


endif // i_utgHH
```

23 USING ARM RULES FOR NAME ENCODING "OPERATOR=" BECOMES: "OPER_AS"

"operator+=" ⟶ operator+= ⟶ OPER_APL

"operator[]" ⟶ operator[] ⟶ OPER_VC

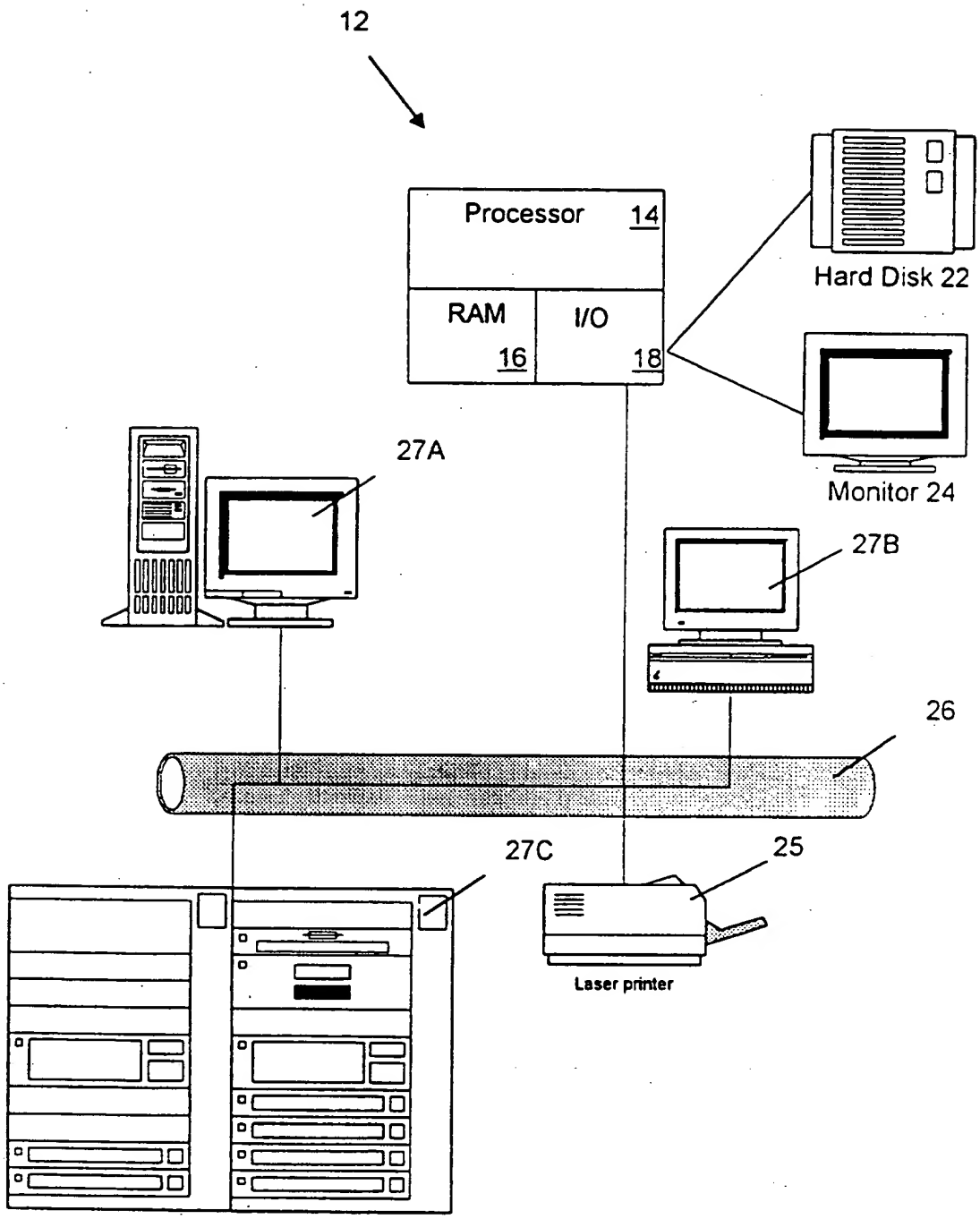

12
Processor 14
RAM 16
I/O 18
Hard Disk 22
Monitor 24
27A
27B
26
27C
25
Laser printer

Figure 3

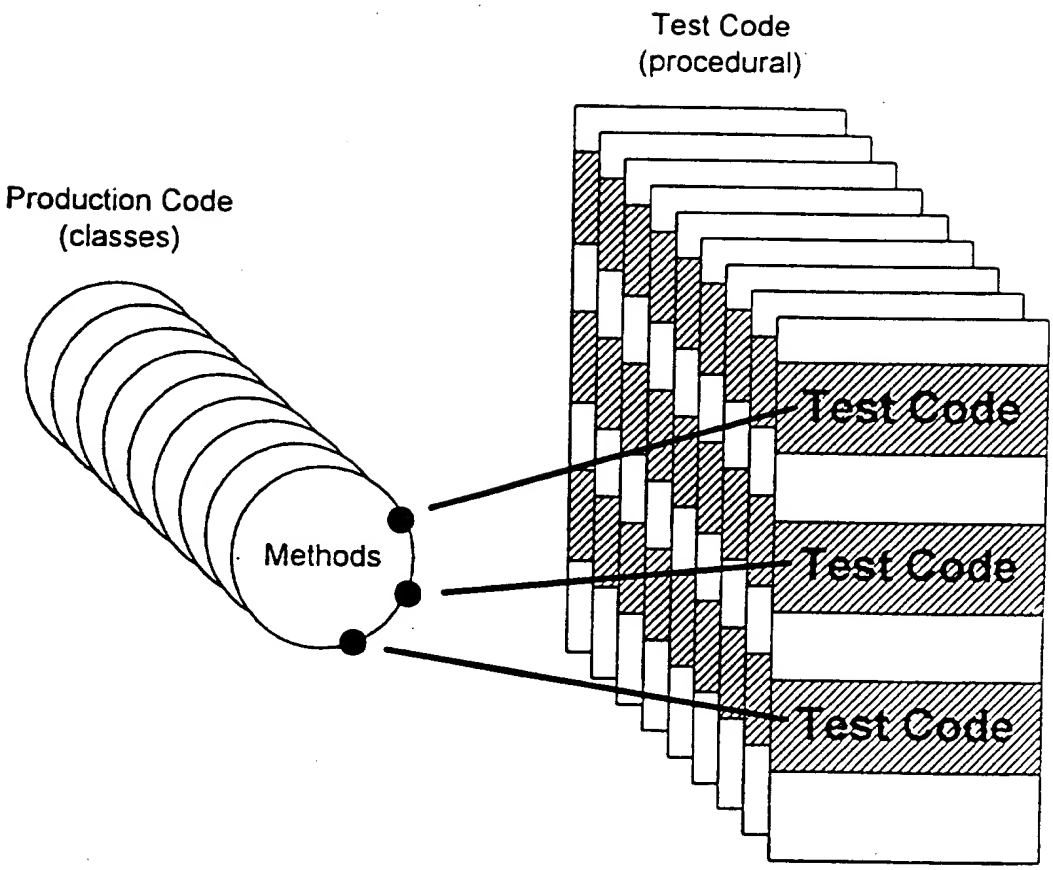


Figure 2
(prior art)

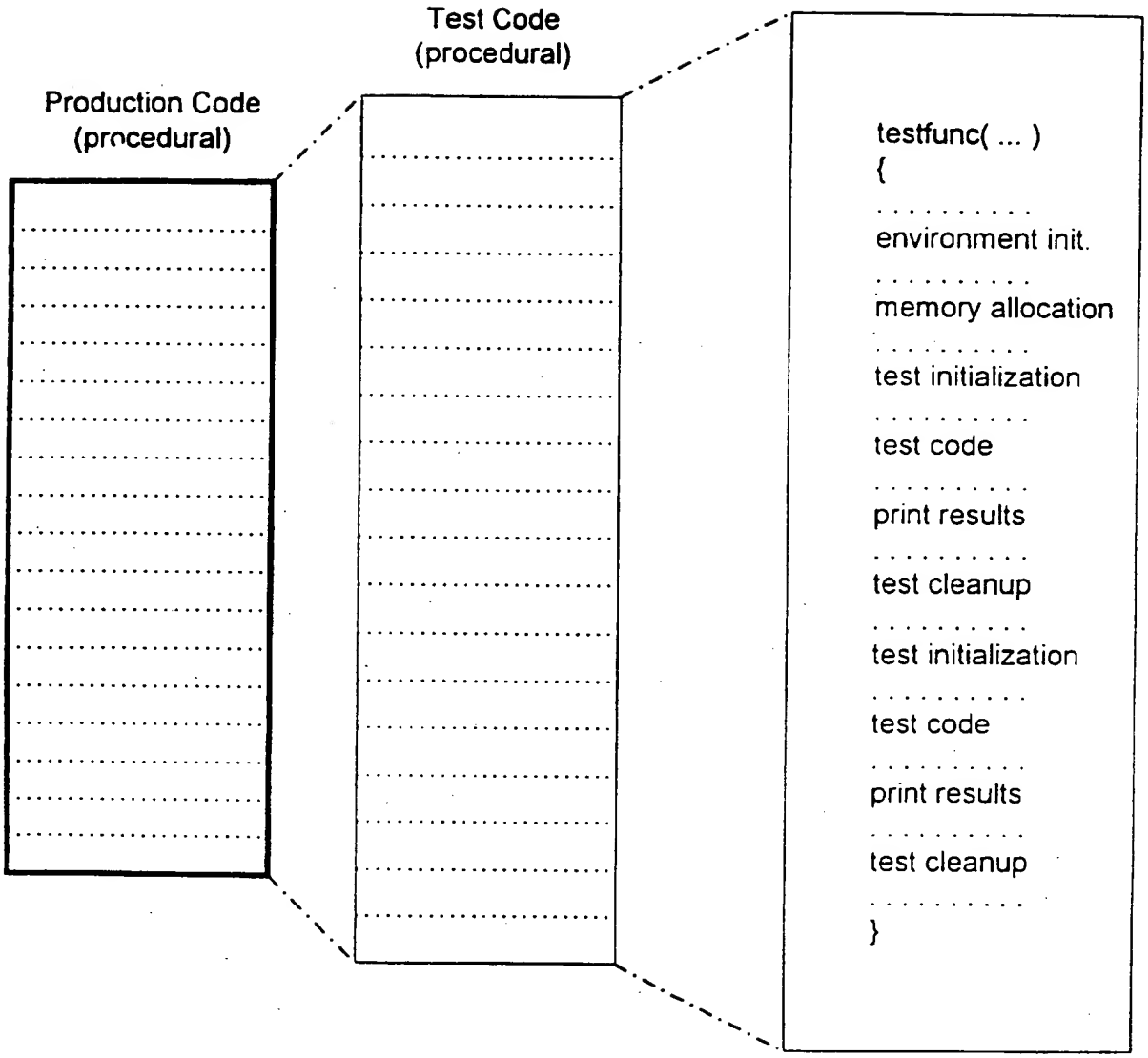


Figure 1
(prior art)

said inspection object execution means includes means for executing a method member of said inspection object for invoking one or more test suite members.

80. An apparatus according to claim 69, wherein

said inspection object execution means includes means for placing said digital data processor in a desired runtime environment.

results of such invocation, and for generating said report signal to be indicative of such comparison.

74. An apparatus according to any of claims 69 and 73, wherein

said inspection object execution means includes means for executing one or more member methods of said inspection object to generate said report signal.

75. An apparatus according to any of claims 69, wherein:

said inspection object execution means includes means responding to a verbosity control signal to generate said report signals with corresponding verbosity.

76. An apparatus according to claim 69, wherein

said inspection object execution means includes means for executing one or more method members of said inspection object, referred to hereinafter as inspection members, for testing corresponding method members of said test object.

77. An apparatus according to claim 76, wherein

said inspection object execution means includes means for executing one or more members of said inspection object, referred to hereinafter as test suite members, for exercising one or more of said inspection members.

78. An apparatus according to claim 77, wherein

said inspection object execution means includes means for executing said test suite members for at least one of

(i) testing for memory leaks in connection with at least one of creation and destruction said test object,

(ii) testing accessor and transformer members of said test object,

(iii) testing operator member methods of said test object,

(iv) testing members involved in persistence of said test object, and

(v) testing method members semantically unique to said test object.

79. An apparatus according to claim 77, wherein

for creating an inspection object instantiating an inspection class, and for generating an inspection object invocation signal for invoking one or members thereof,

(B) inspection object execution means, coupled to said test harness means, for responding to said inspection object invocation signal for

(i) creating said test object,

(ii) generating a test object invocation signal to invoke one or more selected method members of said test object,

(iii) generate a signal, hereinafter referred to as a report signal, reporting an effect of such invocation,

(C) test object execution means, coupled to said inspection object execution means, for responding to said test object invocation signal to execute one or more selected method members thereof.

70. An apparatus according to claim 69, wherein

said inspection object execution means includes means for creating said test object as an instantiation of a test class that comprises any of

(a) said subject class, and

(b) an instantiable class derived from said subject class.

71. An apparatus according to claim 69, wherein

said inspection object execution means includes means for applying at least one argument to one or more of said selected method members of said test object in connection with invocation thereof.

72. An apparatus according to claim 69, wherein

said inspection object execution means includes exception service means preventing exceptions that occur during invocation of said one or more selected method members from discontinuing at least reporting on effects of invocation of said test object.

73. An apparatus according to claim 69, wherein

said inspection object execution means includes means for comparing a result of invocation of said one or more selected method members of said test object with expected

said inspection object execution means includes means for executing one or more members of said inspection object, referred to hereinafter as test suite members, for exercising one or more of said inspection members.

66. An apparatus according to claim 65, wherein

said inspection object execution means includes means for executing said test suite members for at least one of

(i) testing for memory leaks in connection with at least one of creation and destruction of said test object,

(ii) testing accessor and transformer members of said test object,

(iii) testing operator member methods of said test object,

(iv) testing members involved in persistence of said test object, and

(v) testing method members semantically unique to said test object.

67. An apparatus according to claim 65, wherein

said inspection object execution means includes means for executing a method member of said inspection object for invoking one or more test suite members.

68. An apparatus according to claim 57, wherein

said inspection object execution means includes means for placing said digital data processor in a desired runtime environment.

69. Apparatus for testing a subject class in an object-oriented digital data processing system, said apparatus comprising:

(A) test harness means for responding to an inspection signal defining an inspection class having one or more members for

(i) creating a test object as an instantiation of any of said subject class and a class derived therefrom,

(ii) invoking one or more selected method members of said test object,

(iii) generating a signal, hereinafter referred to as a report signal, reporting an effect of such invocation

(a) said subject class, and

(b) an instantiable class derived from said subject class.

59. An apparatus according to claim 57, wherein

said inspection object execution means includes means for applying at least one argument to one or more of said selected method members of said test object in connection with invocation thereof.

60. An apparatus according to claim 57, wherein

said inspection object execution means includes exception service means preventing exceptions that occur during invocation of said one or more selected method members from discontinuing at least reporting on effects of invocation of said test object.

61. An apparatus according to claim 57, wherein

said inspection object execution means includes means for comparing a result of invocation of said one or more selected method members of said test object with expected results of such invocation, and for generating said report signal to be indicative of such comparison.

62. An apparatus according to any of claims 57 and 59, wherein

said inspection object execution means includes means for executing one or more member methods of said inspection object to generate said report signal.

63. An apparatus according to any of claims 61, wherein:

said inspection object execution means includes means responding to a verbosity control signal to generate said report signals with a selected level of verbosity.

64. An apparatus according to claim 57, wherein

said inspection object execution means includes means for executing one or more method members of said inspection object, referred to hereinafter as inspection members, for testing corresponding method members of said test object.

65. An apparatus according to claim 64, wherein

said code generator means includes means for generating said inspection signal to define said inspection class to include one or more method members, referred to hereinafter as test suite members, for exercising one or more of said inspection members.

54. An apparatus according to claim 53, wherein

said code generator means includes means for generating said inspection signal to define said inspection class to include one or more method members, hereinafter referred to as run members, for invoking said one or more test suite members.

55. An apparatus according to claim 53, wherein

said code generator means includes means for generating said inspection signal to define said inspection class to include one or more method members providing common services for invocation by said test suite members.

56. An apparatus according to claim 53, wherein

said code generator means includes means for generating said inspection signal to define said inspection class to include one or more method members providing common reporting services for invocation by said test suite members.

57. An apparatus according to claim 50, comprising:

(A) inspection object execution means, coupled to said test harness means, for respondir to said inspection object invocation signal for

(i) creating said test object,

(ii) generating a test object invocation signal to invoke one or more selected method members of said test object,

(iii) generate a signal, hereinafter referred to as a report signal, reporting an effect of suc invocation,

(B) test object execution means, coupled to said inspection object execution means, for responding to said test object invocation signal to execute one or more selected method members thereof.

58. An apparatus according to claim 57, wherein

said inspection object execution means includes means for creating said test object an instantiation of a test class that comprises any of

responding to a verbosity control signal to generate said report signal with a selected level of verbosity.

50. Apparatus for testing a subject class in an object-oriented digital data processing system, said apparatus comprising:

(A) code generator means for responding to a source signal defining said subject class to generate an inspection signal defining an inspection class having one or more members for

(i) creating a test object as an instantiation of any of said subject class and a class derived therefrom,

(ii) invoking one or more selected method members of said test object,

(iii) generate a signal, hereinafter referred to as a report signal, reporting an effect of such invocation, and

(B) test harness means, coupled to said code generator means, for responding to said inspection signal for creating an inspection object instantiating said inspection class, and for generating an inspection object invocation signal for invoking one or more members thereof.

51. An apparatus according to claim 50, wherein

said code generator means includes test class generating means for responding to said source signal to generate a test signal defining a test class that comprises any of

(i) said subject class, and

(ii) an instantiable class derived from said subject class, and wherein

said code generator further includes means for generating said inspection signal to define said inspection class to include one or more members for creating said test object as an instantiation of said test class.

52. An apparatus according to claim 50, wherein

said code generator means includes means for generating said inspection signal to define said inspection class to include one or more method members, referred to hereinafter as inspection members, for testing corresponding method members of said test object.

53. An apparatus according to claim 52, wherein

step (B)(ii) includes the step of invoking one or more method members of said inspection object, referred to hereinafter as inspection members, for testing corresponding method members of said test object.

44. A method according to claim 43, wherein

step (B)(ii) includes the step of invoking one or more method members of said inspection object, referred to hereinafter as test suite members, for exercising one or more of said inspection members.

45. A method according to claim 44, wherein

step (B)(ii) includes the step of invoking one or more test suite members to at least one of

(i) test for memory leaks in connection with at least one of creation and destruction of said test object,

(ii) test accessor and transformer members of said test object,

(iii) test operator member methods of said test object,

(iv) test members involved in persistence of said test object, and

(v) test method members semantically unique to said test object.

46. A method according to claim 44, wherein

step (B)(ii) includes the step invoking one or more method members of said inspection object for invoking one or more test suite members.

47. A method according to claim 34, wherein

step (B)(i) includes the step of placing said digital data processor in a desired runtim environment.

48. A method according to any of claims 38 - 42, comprising:

invoking one or more method members of said inspection object to generate said report signal.

49. A method according to any of claims 48, comprising:

36. A method according to claim 34, wherein

step (B)(ii) includes applying at least one argument to one or more of said selected method members of said test object in connection with invocation thereof.

37. A method according to claim 36, wherein

step (B)(ii) includes applying at least one argument to a constructor member of said test object in connection with creation of said test object.

38. A method according to claim 34, wherein

step (B)(ii) includes the step of preventing exceptions that occur during invocation of said one or more selected method members from discontinuing execution of steps (B)(ii) and (B)(iii).

39. A method according to claim 34, wherein

step (B)(iii) includes comparing said result of such invocation with one or more expected values thereof, and generating said report signal to be indicative of such comparison.

40. A method according to claim 39, wherein

step (B)(iii) includes the step of storing a signal indicative of said effects in a data member of said inspection object.

41. A method according to claim 34, wherein

said method includes the step of executing step (B) a plurality of times, each for invoking a different group of one or more selected method members of said test object.

42. A method according to claim 41, wherein

step (B)(iii) includes the step of generating said report signal to be indicative of at least one of

(i) comparison of results of invocation of one or more method members of said test object with one or more expected values thereof, and

(ii) detection of a memory leak.

43. A method according to claim 34, wherein

step (B)(i) includes the step of placing said digital data processor in a mode of a desired runtime environment.

32. A method according to any of claims 20 - 25, comprising:

invoking one or more method members of said inspection object to generate said report signal.

33. A method according to any of claims 32, comprising:

responding to a verbosity control signal to generate report signals with a selected lev of verbosity.

34. A method for testing a subject class in an object-oriented digital data processing system, said method comprising:

(A) creating an inspection object as an instantiation of an inspection class having one or more members for

(i) creating a test object as an instantiation of any of said subject class and a class deriv therefrom,

(ii) invoking one or more selected method members of said test object,

(iii) generating a signal, hereinafter referred to as a report signal, reporting an effect of such invocation

(B) invoking one or more members of said inspection object to

(i) create said test object,

(ii) invoke one or more selected method members of said test object,

(iii) generate a signal, hereinafter referred to as a report signal, reporting an effect of suc invocation.

35. A method according to claim 34, wherein

step (B)(i) includes the step of creating said test object as an instantiation of a test class that comprises any of

(a) said subject class, and

(b) an instantiable class derived from said subject class.

(i) a comparison of results of invocation of one or more method members of said test object with one or more expected values thereof, and

(ii) detection of a memory leak.

26. A method according to claim 17, wherein

said method includes the step of executing step (B) a plurality of times, each for invoking a different group of one or more selected method members of said test object.

27. A method according to claim 17, wherein

step (C)(ii) includes the step of invoking one or more method members of said inspection object, referred to hereinafter as inspection members, for testing corresponding method members of said test object.

28. A method according to claim 27, wherein

step (C)(ii) includes the step of invoking one or more method members one or more members of said inspection object, referred to hereinafter as test suite members, for exercising one or more of said inspection members.

29. A method according to claim 28, wherein

step (C)(ii) includes the step of invoking one or more test suite members to at least one of

(i) test for memory leaks in connection with at least one of creation and destruction of said test object,

(ii) test accessor and transformer members of said test object,

(iii) test operator member methods of said test object,

(iv) test members involved in persistence of said test object, and

(v) test method members semantically unique to said test object.

30. A method according to claim 28, wherein

step (C)(ii) includes the step invoking one or more method members of said inspection object for invoking one or more test suite members.

31. A method according to claim 17, wherein

18. A method according to claim 17, wherein

step (C)(ii) includes applying at least one argument to one or more of said selected method members of said test object in connection with invocation thereof.

19. A method according to claim 18, wherein

step (C)(i) includes applying at least one argument to a constructor member of said test object in connection with creation thereof.

20. A method according to claim 17, wherein

step (C)(ii) includes the step of preventing exceptions that occur during invocation of said one or more selected method members from discontinuing execution of steps (C)(ii) and C(iii).

21. A method according to claim 17, wherein

step (C)(iii) includes comparing said result of such invocation with one or more expected values thereof, and generating said report signal to be indicative of such comparison.

22. A method according to claim 17, wherein

step (C)(iii) includes the step of storing said report signal in a data member of said inspection object.

23. A method according to claim 22, wherein

step (C)(iii) includes the step of storing in said data member of said inspection object a signal indicative of a number of errors incurred in connection with invocation of said one or more selected method members of said test object.

24. A method according to claim 23, wherein

step (C)(iii) includes the step of storing in said data member of said inspection object a signal indicating coverage of testing associated with invocation of said one or more method members of said test object.

25. A method according to claim 22, wherein

step (C)(iii) includes the step of generating said report signal to be indicative of at least one of

step (A) includes the step of generating said inspection signal to define said inspection class to comprise one or more method members providing common reporting services for invocation by said test suite members.

13. A method according to claim 7, wherein

step (A) includes the step of generating said inspection signal to define said inspection members to include method member functions corresponding to at least one constructor in said test object, wherein each of those method member functions take substantially the same arguments as the corresponding constructor in said test object.

14. A method according to claim 7, wherein

step (A) includes the step of generating said inspection signal to define said inspection members to include method member functions corresponding to a destructor in said test object.

15. A method according to claim 7, wherein

step (A) includes the step of generating said inspection signal to define said inspection members to include method member functions corresponding to at least one operator function in said test object.

16. A method according to claim 7, wherein

step (A) includes the step of generating said inspection signal to define said inspection members to have function names similar to those of the corresponding method members of the test object that they test.

17. A method according to claim 1, comprising the step of

(B) responding to said inspection signal to create an inspection object as an instantiation of said inspection class, and

(C) invoking one or more members of said inspection object to

(i) create said test object,

(ii) invoke one or more selected method members of said test object,

(iii) generate a signal, hereinafter referred to as a report signal, reporting an effect of such invocation.

step (B) includes the step of generating said test signal to define said test class to giv instantiations of said inspection class access to members of said subject class.

7. A method according to claim 1, wherein

step (A) includes the step of generating said inspection signal to define said inspectic class to include one or more method members, referred to hereinafter as inspection member for testing corresponding method members of said test object.

8. A method according to claim 7, wherein

step (A) includes the step of generating said inspection signal to define said inspectic class to include one or more method members, referred to hereinafter as test suite members. for exercising one or more of said inspection members.

9. A method according to claim 8, wherein

step (A) includes the step of generating said inspection signal to define said test suit members to include one or more members for at least one of

(i) testing for memory leaks in connection with at least one of creation and destruction said test object,

(ii) testing accessor and transformer members of said test object,

(iii) testing operator member methods of said test object,

(iv) testing members involved in persistence of said test object, and

(v) testing method members semantically unique to said test object.

10. A method according to claim 8, wherein

step (A) includes the step of generating said inspection signal to define said inspecti class to include one or more method members, hereinafter referred to as test run members. invoking said one or more test suite members.

11. A method according to claim 8, wherein

step (A) includes the step of generating said inspection signal to define said inspect class to comprise one or more method members providing common services for invocatio by said test suite members.

12. A method according to claim 8, wherein

CLAIMS:

1. A method for testing a subject class in an object-oriented digital data processing system, said method comprising the step of:

(A) responding to a source signal defining said subject class to generate an inspection signal defining an inspection class having one or more members for

(i) creating a test object as an instantiation of any of said subject class and a class derived therefrom,

(ii) invoking one or more selected method members of said test object,

(iii) generating a signal, hereinafter referred to as a report signal, reporting an effect of such invocation.

2. A method according to claim 1, comprising

(B) responding to said source signal to generate a test signal defining a test class that comprises any of

(i) said subject class, and

(ii) an instantiable class derived from said subject class, and

step (A)(i) includes the step of generating said inspection signal to define said inspection class to include one or more members for creating said test object as an instantiation of said test class.

3. A method according to claim 2, wherein

step (B) includes the step of generating said test signal to define an instantiable test class that inherits one or more members of said subject class.

4. A method according to claim 3, wherein

step (B) includes the step of generating said test signal to define said test class to substantially duplicate pure virtual functions of said subject class, absent constructs that denote those functions as having both pure and virtual attributes.

5. A method according to claim 3, wherein

step (B) includes the step of generating said test signal to define said test class to substantially duplicate at least one constructor and a destructor of said subject class.

6. A method according to claim 3, wherein

```
 METHOD START: t_LifeCycle
*****************************************
Starting test: Ctor1 and Dtor, at line: 56, File: c:\view\prodn\fd\errors\ut\insp\erMsg1.cpp
 METHOD START: newInstance
 METHOD END: newInstance
 OBJECT:
  - erMsg *
    * bcString18n *
     Value:
        * csm_character *
        Value: erMsg1 t_LifeCycle:\view\prodn\fd\errors\ut\insp\erMsg1.cpp
        Number of Chars: 65
        - csm_character -
     Advanced: 0
    - bcString18n -
  : False
    * bcString18n *
     Value: NULL
     Advanced: 0
    - bcString18n -
  - erMsg -
 METHOD START: deleteInstance
 METHOD END: deleteInstance
ERROR: MEMORY LEAK DETECTED IN MEMORY MANAGER, in test: Ctor1 and Dtor, at Line: 66, File: c:\view\prodn\...
 FAILED test: Ctor1 and Dtor, at line: 66, File: c:\view\prodn\fd\errors\ut\insp\erMsg1.cpp
*****************************************
```

APPENDIX N

```
----------------------------------------
 Starting test: Copy Ctor - Copying Ctor1 across Memory Mgrs, at line: 133, File: c:\view\prodn\fd\...
ERROR: MEMORY LEAK DETECTED IN MEMORY MANAGER, in test: Copy Ctor - Copying Ctor1 across Memory Mgrs, at Line: 146
ERROR: MEMORY LEAK DETECTED IN MEMORY MANAGER, in test: Copy Ctor - Copying Ctor1 across Memory Mgrs, at Line: 146
 FAILED test: Copy Ctor - Copying Ctor1 across Memory Mgrs, at line: 146, File: c:\view\prodn\fd\...
----------------------------------------
...
----------------------------------------
Starting test: qryMsgID, at line: 268, File: c:\view\prodn\fd\errors\ut\insp\erMsgi.cpp
 CHECKED: Expression: qryMsgID( msg1 ) == 42, in test: qryMsgID, at line: 268, File: c:\view\prodn\fd\...
 PASS test: qryMsgID, at line: 268, File: c:\view\prodn\fd\errors\ut\insp\erMsgi.cpp
----------------------------------------
```

APPENDIX M

```
ERROR: MEMORY LEAK DETECTED IN MEMORY MANAGER, in test: Ctor1 and Dtor, at Line: 66, File: c:\view\...
```

APPENDIX L

63

```
TESTRUNdeclare(erMsgi)

extern "C" {
        dtChar * FAR PASCAL EXPORT testname()
    {
        return( "erMsgi" );
    }

    dtInt FAR PASCAL EXPORT testmain( idtTestArgs& aTestArgs )
    {
    return( TESTRUN(erMsgi)( aTestArgs ) );
    }
}       // end extern C
```

APPENDIX K

I_UTG.CPP

```
////////////////////////////////////////////////////////////////////////
//
// FILE_NAME: i_utg.cpp
//
// Copyright 1993 by Marcam Corp., Newton, MA USA
//
// This unpublished copyrighted work contains
// TRADE SECRET information of Marcam Corporation.
//
// Use, transfer, disclosure, or copying without
// its expressed written permission is strictly
// forbidden.
//
//
////////////////////////////////////////////////////////////////////////

#include "i_utg.hh"            // For i_utg

// Establish an instance of TESTRUN for this class
TESTRUNdeclare(i_utg)

//
// TESTRUN() and testname() are used by the unit test harness
// to run the tests and display the name of the unit test.
//
extern "C" dtInt FAR PASCAL EXPORT testmain( idtTestArgs& aTestArgs )
{
    return( TESTRUN(i_utg)( aTestArgs ) );
}

extern "C" dtChar * FAR PASCAL EXPORT testname()
{
    return( "i_utg" );
}

////////////////////////////////////////////////////////////////////////
// !CLASS_DESCRIBER
// !LIBRARY
// !NAME i_utg
// !TEXT
// Generated 7/12/93 by utg version 1.0
//
// !AUTHOR      <Author's name>   <(Author's E-mail address)>
// !REVIEWER    <Reviewer's name> <(Reviewer's E-mail address)>
// !REVIEW_DATE <date>
////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////
// Inspection class constructor.
//////////////////////////////////////////////////

i_utg::i_utg(
    dtStreamExecutive* aStreamExec,
    dtMemoryExecutive& aMemExec,
    idtTestArgs&       aTestArgs,
    dtMemoryMgr*
    ): idtTest( aStreamExec, aMemExec, aTestArgs )

{
    for ( int i = 0 ; i < TOTAL_FUNCS ; i++ )
        coverage[i] = 0;
}

//////////////////////////////////////////////////
// Inspection class destructor.
```


APPENDIX C

46

I_UTG.CPP

```
////////////////////////////////////////////////

i_utg::~i_utg()
{
}


////////////////////////////////////////////////
// Lifecycle testing.
//
// The TLIFE_START and TLIFE_END macros are placed
// around each lifecycle test.  These provide the
// announcement of each test and whether it has
// passed or failed.  They also provide clean
// memory mamangers for each test.  Throughout
// the lifecycle test (between the TLIFE_INIT and
// TLIFE_CLEANUP macros) two memory managers are
// available: mm and mm2.
////////////////////////////////////////////////

dtVoid
i_utg::t_LifeCycle()
{
    TLIFE_INIT

    // CONSTRUCTOR/DESTRUCTOR TESTING
    //
    // Constructor and destructor testing.  One test must
    // be performed for each constructor.  The basic format
    // is construct the object, announce it, then delete it.
    //
    TLIFE_START( "ctor1 and dtor" )
        s_utg* testobjX = newInstance( /*PARAMETERS,*/ mm );
        announceObject( *testobjX );
        deleteInstance( testobjX );
    TLIFE_END


    // COPY CONSTRUCTOR TESTING
    //
    // Copy constructor tests.  One test must be performed
    // for each copy constructor/constructor combination.
    // The format for each test is: construct an object
    // using mm, then invoked the copy constructor using
    // mm2, delete the original, announce the copy, then
    // delete the copy.  This test copying across memory
    // spaces, as well as, incomplete copies and use of
    // aliases (shared pointers with no reference counting).
    //
    TLIFE_START( "Copy ctor and ctor1" )
        s_utg* testobjX = newInstance( /*PARAMETERS,*/ mm );
        s_utg* testobjY = copyInstance( testobjX, mm2 );
        deleteInstance( testobjX );
        announceObject( *testobjY );
        deleteInstance( testobjY );
    TLIFE_END


    // ASSIGNMENT OPERATOR TESTING
    //
    // Similar in nature to the copy constructor testing,
    // one test must be performed for each constructor/
    // assignment operator combination.  The format is:
    // construct objectX using mm, construct objectY
    // using mm2, assign objectX to objectY, check for
    // equality if that operator is defined, delete
    // objectX, announce objectY, delete objectY.
    // Additionally, chained assignment (objZ=objY=objX,
    // where after the operation objZ == objX) and
    // assignment to self (objX=objX, where objX still
```

11 12 13 14 15 16 17 18

47

I_UTG.CPP

```
    // has a value after assignment) must be tested.
    //
    TLIFE_START( "ctor1 and operator=" )
        s_utg* testobjX = newInstance( /*PARAMETERS,*/ mm );
        s_utg* testobjY = newInstance( /*PARAMETERS,*/ mm2 );
        *testobjY = *testobjX;
//!FIX        checkExpr( *testobjX == *testobjY, currTest );
        deleteInstance( testobjX );
        announceObject( *testobjY );
        deleteInstance( testobjY );
    TLIFE_END


    TLIFE_START( "Chained assignment" )
        s_utg* testobjX = newInstance( /*PARAMETERS,*/ mm );
        s_utg* testobjY = newInstance( /*PARAMETERS,*/ mm2 );
        s_utg* testobjZ = newInstance( /*PARAMETERS,*/ mm );
        *testobjZ = *testobjY = *testobjX;
//!FIX        checkExpr( *testobjZ == *testobjY, currTest );
        deleteInstance( testobjX );
        deleteInstance( testobjY );
        announceObject( *testobjZ );
        deleteInstance( testobjZ );
    TLIFE_END


    TLIFE_START( "Assignment to self" )
        s_utg* testobjX = newInstance( /*PARAMETERS,*/ mm );
        *testobjX = *testobjX;
        announceObject( *testobjX );
        deleteInstance( testobjX );
    TLIFE_END


    TLIFE_CLEANUP;
}

////////////////////////////////////////////////
// Operators testing.
//
// All operators are tested here.  The entire
// set of operators has been broken into like
// categories for ease of testing.  Between the
// T_INIT and T_CLEANUP macros the memory manager
// variable mm is available for constructing
// test objects.  Each individual test must be
// surrounded by the T_START and T_END macros,
// or alternately, the test itself can be performed
// within a TEST macro.
////////////////////////////////////////////////

dtVoid
i_utg::t_Operators()
{
    T_INIT( "t_Operators" )

    // COMPARISON OPERATORS: < > <= >= == !=

    // ARITHMETIC OPERATORS: + - * / % ++ --

    // LOGICAL OPERATORS: || && !

    // BITWISE OPERATORS: ^ | & ~ << >>

    // EXTENDED ASSIGNMENT OPERATORS: += -= *= /= %= ^= |= &= <<= >>=

    // CONVERSION OPERATORS:  type()
```

8/18/93 12:51AM Page 3

```
    // MISCELLANEOUS AND GLOBAL OPERATORS

    T_CLEANUP
}

////////////////////////////////////////////////
// Set and Query testing.
//
// The set and query member functions are tested
// here.  If your class under test contains
// values that do not require being set first,
// then test the query methods for those values
// before testing the corresponding set methods.
// Between the T_INIT and T_CLEANUP macros the
// memory manager variable mm is available for
// constructing test objects.  Each individual
// test must be surrounded by the T_START and T_END
// macros, or alternately, the test itself can be
// performed within a TEST macro.
////////////////////////////////////////////////

dtVoid
i_utg::t_SetQrys()
{
    T_INIT( "t_SetQrys" )

    // QUERY TESTS


    // SET TESTS


    T_CLEANUP
}

////////////////////////////////////////////////
// Semantics testing.
//
// These tests are for those member functions
// unique to the production class.
// Between the T_INIT and T_CLEANUP macros the
// memory manager variable mm is available for
// constructing test objects.  Each individual
// test must be surrounded by the T_START and T_END
// macros, or alternately, the test itself can be
// performed within a TEST macro.
////////////////////////////////////////////////

dtVoid
i_utg::t_Semantics()
{
    T_INIT( "t_Semantics" )

    //!FIX Add your semantic member function tests here.


    T_CLEANUP;
}

////////////////////////////////////////////////
// Persistence testing.
//
// This tests behavior specific to the persistence
// member functions of a production class.
// Between the T_INIT and T_CLEANUP macros the
// memory manager variable mm is available for
```

49

I_UTG.CPP

```
// constructing test objects.  Each individual
// test must be surrounded by the T_START and T_END
// macros, or alternately, the test itself can be
// performed within a TEST macro.
////////////////////////////////////////////////////

dtVoid
i_utg::t_Persist()
{
    T_INIT( "t_Persist" )

    //!FIX Add your persistent member function tests here.


    T_CLEANUP;
}


////////////////////////////////////////////////////
// Return coverage information.
//
// NOTE: This is not to be modified by unit
//       test developers.
////////////////////////////////////////////////////

dtVoid
i_utg::qryCoverage(
    dtInt*& aCovAry,
    dtInt&  aNumFuncs
    ) const
{
    aCovAry = coverage;
    aNumFuncs = TOTAL_FUNCS;
}


////////////////////////////////////////////////////
// Constructors for class under inspection.
//
// NOTE: The following constructors can
//       be modified by unit test developers.
//       The parameters for a given constructor,
//       and the announcement of parameters and
//       resulting objects can be customized
//       by the unit test developer.
////////////////////////////////////////////////////

s_utg*
i_utg::newInstance(
    dtMemoryMgr* aMemMgr
    )
{
    coverage[f_newInstance1]++;
    announceMethodStart( "newInstance1" );
    s_utg* NewObj = new (aMemMgr) s_utg(aMemMgr);
    announceMethodEnd( "newInstance1" );
    return( NewObj );
}

s_utg*
i_utg::newInstance(
    dtULong             aArg1,
    const dtChar* const aArg2,
    dtMemoryMgr*        aMemMgr )
{
    coverage[f_newInstance2]++;
    announceMethodStart( "newInstance2" );
    s_utg* NewObj = new (aMemMgr) s_utg( aArg1, aArg2, aMemMgr );
```

30 31 32 33 34 35 36 37 38

```
    announceMethodEnd( "newInstance2" );
    return( NewObj );
}


s_utg*
i_utg::newInstance(
    dtULong             aArg1,
    const dtChar* const aArg2,
    dtInt               aArg3,
    dtMemoryMgr*        aMemMgr )
{
    coverage[f_newInstance3]++;
    announceMethodStart( "newInstance3" );
    s_utg* NewObj = new (aMemMgr) s_utg( aArg1, aArg2, aArg3, aMemMgr );
    announceMethodEnd( "newInstance3" );
    return( NewObj );
}


////////////////////////////////////////////////
// Copy constructor for class under inspection.
//
// NOTE: The following copy constructor can be
//       modified by unit test developers.
////////////////////////////////////////////////

s_utg*
i_utg::copyInstance(
    s_utg* aInstance,
    dtMemoryMgr* aMemMgr
    )
{
    coverage[f_copyInstance]++;
    announceMethodStart( "copyInstance" );
    s_utg* NewObj = new (aMemMgr) s_utg( *aInstance, aMemMgr );
    announceMethodEnd( "copyInstance" );
    return( NewObj );
}


////////////////////////////////////////////////
// Destructor for class under inspection.
//
// NOTE: The following destructor can be
//       modified by unit test developers.
////////////////////////////////////////////////

dtVoid
i_utg::deleteInstance( s_utg* aInstance )
{
    coverage[f_deleteInstance]++;
    announceMethodStart( "deleteInstance" );
    delete aInstance;
    announceMethodEnd( "deleteInstance" );
}

////////////////////////////////////////////////
// Other member functions.
//
// NOTE: The following inspection functions can be
//       modified by unit test developers.
////////////////////////////////////////////////

s_utg&
i_utg::oper_equal(
    s_utg* aInstance,
    const s_utg& aRhs
    )
{
```

39

40

41

```
    coverage[f_oper_equal]++;
    announceMethodStart( "oper_equal" );
    announceParameter( aRhs );
    aInstance->operator=( *aInstance );
    announceMethodEnd( "oper_equal" );
    return( *aInstance );
}


dtULong
i_utg::oper_dtULong(
    s_utg* aInstance
    )
{
    coverage[f_oper_dtULong]++;
    announceMethodStart( "oper_dtULong" );
    dtULong testVal = aInstance->operator dtULong();
    announceMethodEnd( "oper_dtULong" );
    return( testVal );
}

s_utg&
i_utg::oper_plusequal(
    s_utg* aInstance,
    dtInt aRhs
    )
{
    coverage[f_oper_plusequal]++;
    announceMethodStart( "oper_plusequal" );
//    announceParameter( aRhs );
    aInstance->operator+=( *aInstance );
    announceMethodEnd( "oper_plusequal" );
    return( *aInstance );
}


dtChar
i_utg::oper_index(
    s_utg* aInstance,
    dtInt aIndex
    )
{
    coverage[f_oper_index]++;
    announceMethodStart( "oper_index" );
//  announceParameter( aIndex );
    dtChar testVal = aInstance->operator[](aIndex);
    announceMethodEnd( "oper_index" );
    return( testVal );
}


dtBoolean
i_utg::setID(
    s_utg* aInstance,
    const dtULong aID,
    const dtChar* const aName
    )
{
    coverage[f_setID]++;
    announceMethodStart( "setID" );
//  announceParameter( aID );
//  announceParameter( aName );
    dtBoolean testVal = aInstance->setID(aID,aName);
    announceMethodEnd( "setID" );
    return( testVal );
}
```

```
erStatus
i_utg::openForRead(
    s_utg*& aArg,
    dtMemoryMgr* aMM
    )
{
    coverage[f_openForRead]++;
    announceMethodStart( "openForRead" );
    erStatus testVal = utg::openForRead( (utg*&)aArg, aMM );
    announceRetVal( testVal );
    announceMethodEnd( "openForRead" );
    return( testVal );
}


////////////////////////////////////////////////
// Nonclass functions to be tested.
//
// NOTE: The following nonclass inspection functions
//       can be modified by unit test developers.
////////////////////////////////////////////////


dtBoolean
i_utg_export(
    idtTest* testObj,
    const s_utg& aUtg
    )
{
    coverage[f_i_utg_export]++;
    testObj->announceMethodStart( "i_utg_export" );
    testObj->announceParameter( aUtg );
    dtBoolean testVal = export( aUtg );
    testObj->announceMethodEnd( "i_utg_export" );
    return( testVal );
}


s_utg&
i_utg_oper_plus(
    idtTest* testObj,
    const s_utg& aLhs,
    const s_utg& aRhs
    )
{
    coverage[f_i_utg_oper_plus]++;
    testObj->announceMethodStart( "i_utg_oper_plus" );
    testObj->announceParameter( aLhs );
    testObj->announceParameter( aRhs );
    s_utg& testVal = (s_utg&)operator+( aLhs, aRhs );
    testObj->announceMethodEnd( "i_utg_oper_plus" );
    return( testVal );
}
```

46

```
////////////////////////////////////////////////////////////////////////////
//
// FILE_NAME:   basresc.hh
//
// Copyright 1993 by Marcam Corp., Newton, MA USA
//
// This unpublished copyrighted work contains
// TRADE SECRET information of Marcam Corporation.
//
// Use, transfer, disclosure, or copying without its expressed
// written permission is strictly forbidden.
//
////////////////////////////////////////////////////////////////////////////

#ifndef basrescHH
#define basrescHH

#include "geninst.hh"        // For fdGenInst
#include "bcdatet.hh"        // For bcDateTime
#include "nameuk.hh"         // For fdNameUK
#include "dtbitv.hh"         // For dtBitVector (dummy)

class EXPORT BasicResource;


#ifdef __OSE_TEMPLATES__
#include "orb.hh"
#endif

#ifndef SCHEMACOMPILER
#include "basresc.h"
#endif


////////////////////////////////////////////////////////////////////////////
// !CLASS_DECL_S
// !LIBRARY
// !NAME BasicResource
// !TEXT
// BasicRecource is the implementation of a generic instance-derived
// class.  This is an independently persistable kind (IPK).
//
// The user's access rights will be checked whenever fdSecurityContext
// appears as a parameter.  If the user does not have appropriate access
// in the standard UI/App API, an error message will be returned.
//
// A returned pointer to an erMsg or erMsgList indicates an error
// occurred.  A NULL pointer indicates successful completion of
// the member function.
//
// !AUTHOR      <Author's name>   <(Author's E-mail address)>
// !REVIEWER    <Reviewer's name> <(Reviewer's E-mail address)>
// !REVIEW_DATE <date>
//
////////////////////////////////////////////////////////////////////////////


class EXPORT BasicResource : public fdGenInst
{
  public:

    ////////////////////////////////////////////////
    // !METHOD_DECL_S
    // !NAME close
    // !TEXT
    // This static method closes the current object,
    // releasing all database and memory resources.
    // When successful, the aBasicResource pointer
```

```
// is set to Null.
//
// **Standard UI/App API, implemented in stdgi.cc**
//////////////////////////////////////////////////

static erMsg* close( BasicResource*& aBasicResource,
                     dbTransCtx*&    aDbCtx );

//////////////////////////////////////////////////
// !METHOD_DECL_S
// !NAME create
// !TEXT
// This static method creates a new BasicResource,
// which is returned in aBasicResource.  A Null
// pointer is returned if the object could not be
// created, along with an error message.
//
// **Standard UI/App API, implemented in stdgi.cc**
//////////////////////////////////////////////////

static erMsg* create( BasicResource*&   aBasicResource,
                      dbTransCtx*&      aDbCtx,
                      fdSecurityContext aSecurityCtx,
                      fdInstTypes::type aType,
                      dtMemoryMgr*      aMM );

//////////////////////////////////////////////////
// !METHOD_DECL_S
// !NAME create
// !TEXT
// This static method creates a BasicResource from
// another existing instance.  This is analogous
// to a copy constructor. A Null pointer is returned
// if the object could not be created, along with
// an error message.
//
// **Standard UI/App API, implemented in stdgi.cc**
//////////////////////////////////////////////////

static erMsg* create( BasicResource*&       aNewBasicResource,
                      const BasicResource&  aOldBasicResource,
                      dbTransCtx*&          aDbCtx,
                      fdSecurityContext     aSecurityCtx,
                      fdInstTypes::type     aType,
                      dtMemoryMgr*          aMM );

//////////////////////////////////////////////////
// !METHOD_DECL_S
// !NAME openForChange
// !TEXT
// This static method retrieves an existing BasicResource
// from the database by its user key so that it can be
// modified. If an error occurs the aBasicResource
// pointer will be set to Null, and an error message
// is returned.
//
// **Standard UI/App API, implemented in stdgi.cc**
//////////////////////////////////////////////////

static erMsg* openForChange( BasicResource*&   aBasicResource,
                             const fdNameUK&   aBasicResourceUK,
                             dbTransCtx*&      aDbCtx,
                             fdSecurityContext aSecurityCtx,
                             fdInstTypes::type aType,
                             dtMemoryMgr*      aMM );
```

USER KEY CLASS NAME

```
////////////////////////////////////////////////
// !METHOD_DECL_S
// !NAME openForReview
// !TEXT
// This static method retrieves an existing BasicResource
// from the database by its user key in a nonmodifiable
// mode. If an error occurs the aBasicResource
// pointer will be set to Null, and an error message
// is returned.
//
// **Standard UI/App API, implemented in stdgi.cc**
////////////////////////////////////////////////

static erMsg* openForReview( BasicResource*&   aBasicResource,
                             const fdNameUK&   aBasicResourceUK,
                             dbTransCtx*&      aDbCtx,
                             fdSecurityContext aSecurityCtx,
                             fdInstTypes::type aType,
                             dtMemoryMgr*      aMM );


////////////////////////////////////////////////
// !METHOD_DECL_S
// !NAME remove
// !TEXT
// This static method removes an existing BasicResource
// from the database.  When successful, the
// aBasicResource pointer is Null (the object has been
// deleted from memory).  If an error occurs, e.g.,
// validateToRemove() fails, then an error message
// is returned.
//
// **Standard UI/App API, implemented in stdgi.cc**
////////////////////////////////////////////////

static erMsgList* remove( BasicResource*&   aBasicResource,
                          dbTransCtx*&      aDbCtx,
                          fdSecurityContext aSecurityCtx );


////////////////////////////////////////////////
// !METHOD_DECL_S
// !NAME saveAs
// !TEXT
// This static method saves the current object as a new
// object using the supplied user key and type.  The
// original object (aOrigBasicResource) is closed,
// and the new object (aNewBasicResource) is saved
// to the database and remains in memory.  Validation
// errors during save are reported.
//
// **Standard UI/App API, implemented in stdgi.cc**
////////////////////////////////////////////////

static erMsgList* saveAs( BasicResource*&   aNewBasicResource,
                          const fdNameUK&   aBasicResourceUK,
                          BasicResource*&   aOrigBasicResource,
                          dbTransCtx*&      aDbCtx,
                          fdSecurityContext aSecurityCtx,
                          fdInstTypes::type aType,
                          dtMemoryMgr*      aMM );


////////////////////////////////////////////////
// !METHOD_DECL_S
// !NAME validateToRemove
// !TEXT
// This virtual method (declared pure virtual by fdGenInst)
// performs all validation (referential integrity)
// required before removing this object from the database.
```

USER KEY / CLASS NAME

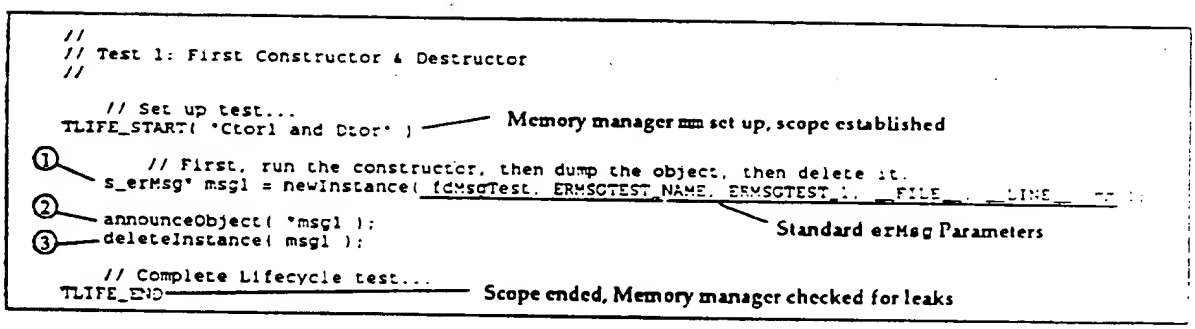
```
class EXPORT i_BasicResource : public idtTest
{
public:

   ////////////////////////////////////////////////
   // Inspection class constructor.
   ////////////////////////////////////////////////
   i_BasicResource(
      dtStreamExecutive* aStreamExec,
      dtMemoryExecutive& aMemExec,
      idtTestArgs&       aTestArgs,
      dtMemoryMgr*       aTestMM );

   ////////////////////////////////////////////////
   // Inspection class destructor.
   ////////////////////////////////////////////////
   ~i_BasicResource();

   ////////////////////////////////////////////////
   // Inspection test suite.
   ////////////////////////////////////////////////
   virtual dtVoid t_LifeCycle();
   virtual dtVoid t_Operators();
   virtual dtVoid t_Semantics();
   virtual dtVoid t_SetQrys();
   virtual dtVoid t_Persist();
   virtual dtVoid qryCoverage( dtInt*& aCovAry, dtInt& aNumFuncs ) const;

   dtVoid setupUKs( fdNameUK&    aUK1,
                    fdNameUK&    aUK2,
                    fdNameUK&    aUK3,
                    dtMemoryMgr* aMM );

   dtVoid setupObj( s_BasicResource* aInstance,
                    dtMemoryMgr*     aMM );

   ////////////////////////////////////////////////
   // Member function close.
   ////////////////////////////////////////////////
   erMsg* close( s_BasicResource*& aBasicResource,
                 dbTransCtx*&      aDbCtx );
```

1 2 3 4

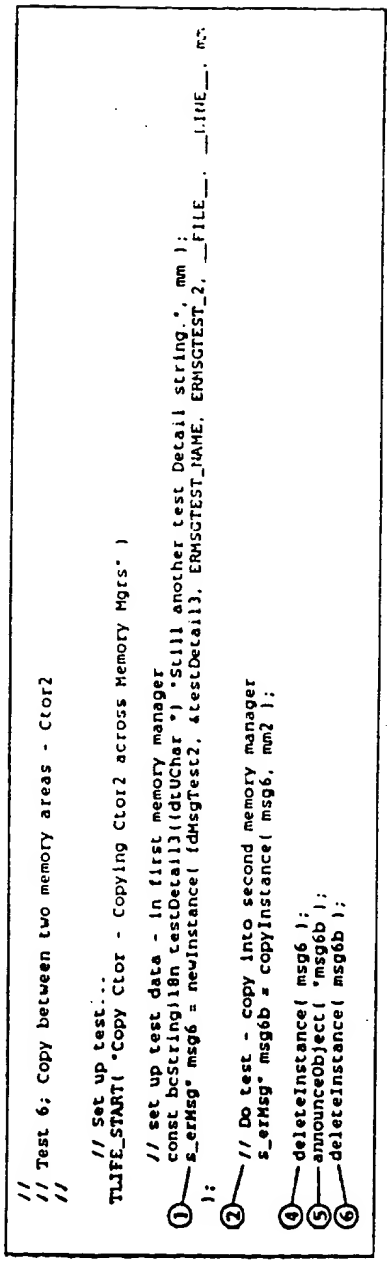
APPENDIX E

```
//////////////////////////////////////////////////
// Persistence testing.
//
// This tests behavior specific to the persistence
// member functions of a production class.
//
// Between the T_INIT and T_CLEANUP macros the
// memory manager variable mm is available for
// constructing test objects.  Each individual
// test must be surrounded by the TEST_START and TEST_END
// macros, or alternately, the test itself can be
// performed within a TEST macro.
//////////////////////////////////////////////////

dtVoid
i_BasicResource::t_Persist()
{
    // STDGIT_TESTCLASS and STDGIT_OBJREF must be established
    //
    #define STDGIT_TESTCLASS s_BasicResource
    #define STDGIT_OBJREF   oObjRef<BasicResource>

    // At least one of the following must be present.
    //
    #define STDGIT_PRODUCTION
    //#define STDGIT_WIP
    //#define STDGIT_TEMPLATE
    //#define STDGIT_IMPORT_EXPORT

    T_INIT( "t_Persist" )

    fdNameUK uk1( mm );
    fdNameUK uk2( mm );
    fdNameUK uk3( mm );
    STDGIT_TESTCLASS* obj1 = (STDGIT_TESTCLASS*) NULL;
    STDGIT_TESTCLASS* obj2 = (STDGIT_TESTCLASS*) NULL;
    STDGIT_TESTCLASS* obj3 = (STDGIT_TESTCLASS*) NULL;
//!FIX  You may need to specify a security context.
    fdSecurityContext security = FD_SC_NO_CONTEXT;
//!FIX  When implemented by STDGIT.CC the KindRef required is named "kr".
//!FIX  fdKindRef< oObjRef<BasicResource>, fdNameUK > kr;

    // Setup the user keys for the 3 test objects.
    //
    setupUKs( uk1, uk2, uk3, mm );

    // Standard test for fdGenInst derived classes.
    // Runs through normal persistent lifecycle for
    // production instances.
    //
    #include "stdgit.cc"

    T_CLEANUP
}

//////////////////////////////////////////////////
// Return coverage information.
//
// NOTE: This is not to be modified by unit
//       test developers.
//////////////////////////////////////////////////
dtVoid
i_BasicResource::qryCoverage(
  dtInt*& aCovAry,
  dtInt&  aNumFuncs
  ) const
{
  aCovAry = coverage;
```

```
    aNumFuncs = TOTAL_FUNCS;
}

////////////////////////////////////////////////
// setupUKs:
//
// Helper function for persistence testing.
// Set up three user keys for the three
// objects used during the STDGIT.
////////////////////////////////////////////////

dtVoid
i_BasicResource::setupUKs(
    fdNameUK1*      aUK1,
    fdNameUK2*      aUK2,
    fdNameUK3*      aUK3,
    dtMemoryMgr*    aMM
    )
{
    // Remember maximum string lengths when setting up test
    // object names and site names.  Names are usually
    // a maximum of 15 characters, sites are 4 characters max.
    //
    bcStringi18n name1( (dtUChar*) "Test Object 111", aMM);
    bcStringi18n name2( (dtUChar*) "Test Object 222", aMM);
    bcStringi18n name3( (dtUChar*) "Test Object 333", aMM);
    bcStringi18n site1( (dtUChar*) "S111", aMM);
    bcStringi18n site2( (dtUChar*) "S222", aMM);
    bcStringi18n site3( (dtUChar*) "S333", aMM);

//!FIX  Set up 3 user keys.  The following works for a basic site
//!FIX  scoped user key (from SNUK skeleton).  Your UK may need more.

    aUK1.setName( name1 );
    aUK1.setSiteName( site1 );

    aUK2.setName( name2 );
    aUK2.setSiteName( site2 );

    aUK3.setName( name3 );
    aUK3.setSiteName( site3 );
}


////////////////////////////////////////////////
// setupObj:
//
// Helper function for persistence testing.
// Set the attributes in the passed object
// so that it is valid and can be saved.
////////////////////////////////////////////////

dtVoid
i_BasicResource::setupObj(
    s_BasicResource* aInstance,
    dtMemoryMgr*     aMM
    )
{
    // Descriptions have a maximum length of 15.
    //
    bcStringi18n descr1( (dtUChar*) "Description 111", aMM);
    bcStringi18n descr2( (dtUChar*) "Description 222", aMM);
    bcStringi18n secGrp( (dtUChar*) "Security GroupX", aMM);

    aInstance->setDescription1( &descr1 );
    aInstance->setDescription2( &descr2 );
    aInstance->setSecurityGroup( &secGrp );

//!FIX  >>> Add additional setup here. <<<
}
```


//
// Test 1: First Constructor & Destructor
//
// Set up test...
TLIFE_START("Ctor1 and Dtor")
Memory manager is set up, scope established
1
// First, run the constructor, then dump the object, then delete it.
s_erMsg* msg1 = newInstance(fcMsgTest, ERMSGTEST_NAME, ERMSGTEST_1, __FILE__, __LINE__);
2
announceObject(*msg1);
3
deleteInstance(msg1);
Standard erMsg Parameters
// Complete Lifecycle test...
TLIFE_END
Scope ended, Memory manager checked for leaks

APPENDIX G


//
// Test 6: Copy between two memory areas - Ctor2
//
// Set up test...
TLIFE_START("Copy Ctor - Copying Ctor2 across Memory Mgrs")
// set up test data - in first memory manager
const bcStringi18n testDetail3((dtUChar *) "Still another test Detail string.", mm);
s_erMsg* msg6 = newInstance(idMsgTest2, &testDetail3, ERMSGTEST_NAME, ERMSGTEST_2, __FILE__, __LINE__, mm
);
// Do test - copy into second memory manager
s_erMsg* msg6b = copyInstance(msg6, mm2);
deleteInstance(msg6);
announceObject("msg6b");
deleteInstance(msg6b);
1
2
4
5
6

APPENDIX H

```
//
// Test 1: qryMsgID
//
s_erMsg* msg1 = newInstance( fdMsgTest3, ERMSGTEST_NAME, ERMSGTEST_3, __FILE__, __LINE__ --

TEST( "qryMsgID", fdMsgTest3 == qryMsgID( msg1 ) )
```

APPENDIX I

```
    // test data
s_erMsg* msg1 = newInstance( fdMsgTest1, ERMSGTEST_NAME, ERMSGTEST_3, __FILE__, __LINE__, mm );
    // set up data for tests with the detail string
const bcString18n testDetail1((dtUChar *) "SetQry test Detail string.", mm );
s_erMsg* msg3 = newInstance( fdMsgTest4, &testDetail, ERMSGTEST_NAME, ERMSGTEST_4, __FILE__, __LINE__, mm
);

//
// Test 5: qryDetail - negative case
//     Since the detail string is NULL,  check that the length is 0
//
TEST( "qryDetail - negative", 0 == qryDetail( msg1 ).qryLength() );

//
// Test 6: qryDetail - positive case
//
TEST( "qryDetail - positive", testDetail == qryDetail( msg1) );
```

APPENDIX J

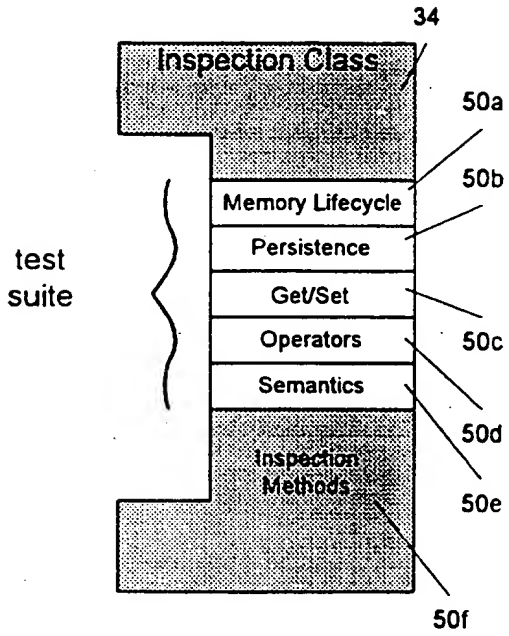

34
Inspection Class
50a
50b
Memory Lifecycle
Persistence
test
suite
Get/Set
Operators
50c
Semantics
50d
Inspection
Methods
50e
50f

Figure 6

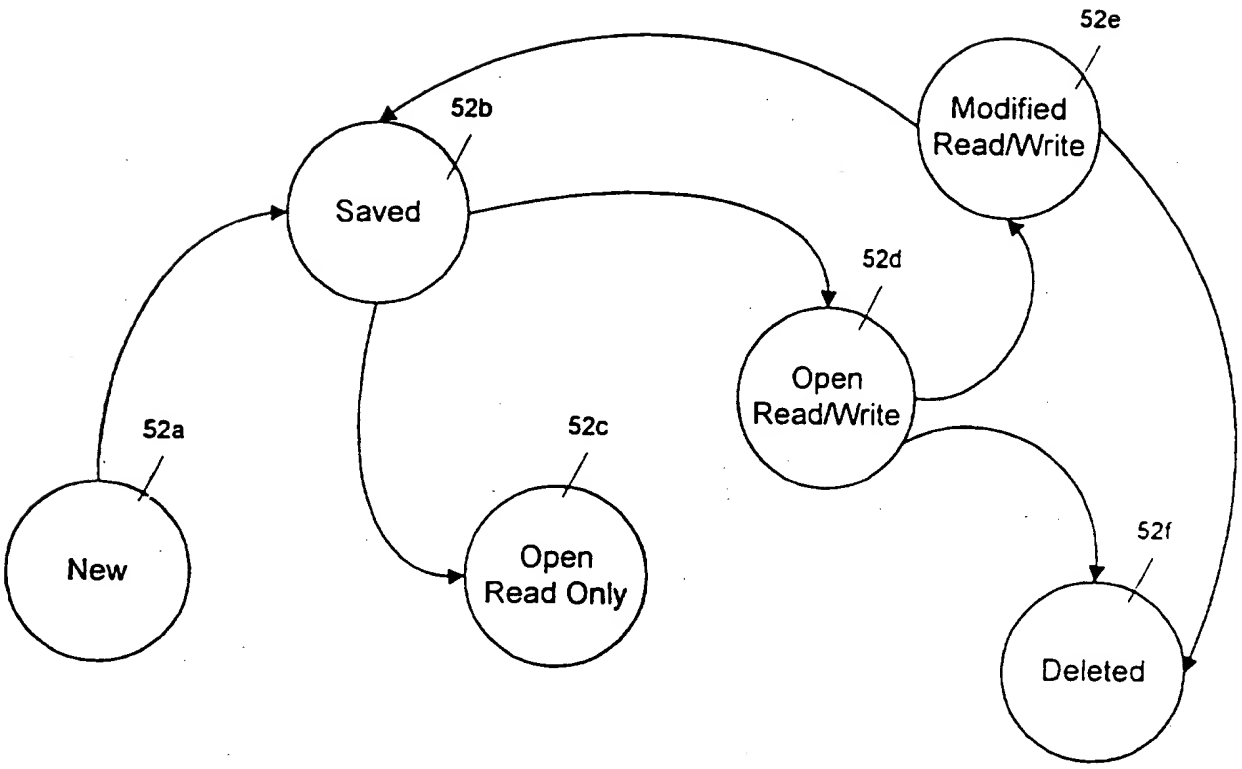

52e
Modified
Read/Write
52b
Saved
52d
Open
Read/Write
52a
New
52c
Open
Read Only
52f
Deleted

Figure 7

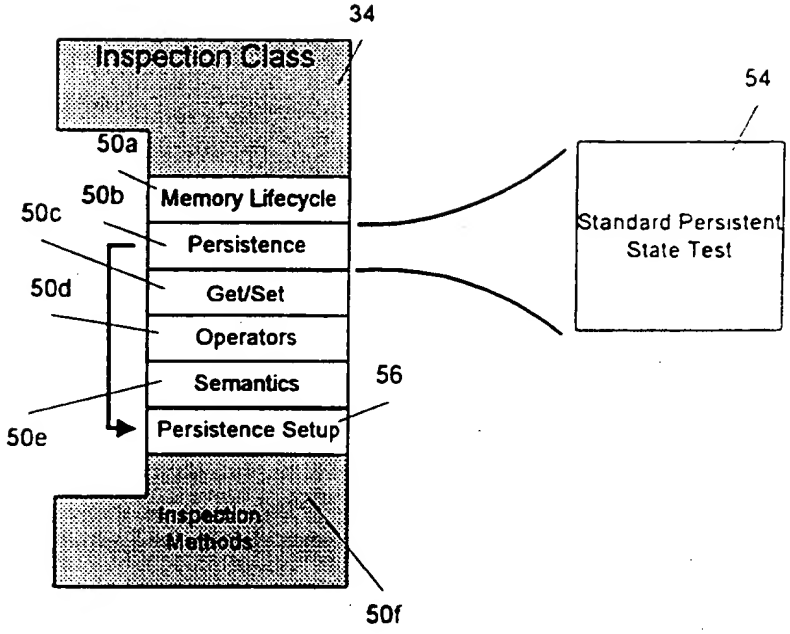

34
Inspection Class
54
50a
50b
Memory Lifecycle
50c
Persistence
Standard Persistent State Test
50d
Get/Set
Operators
Semantics
56
50e
Persistence Setup
Inspection Methods
50f

Figure 8

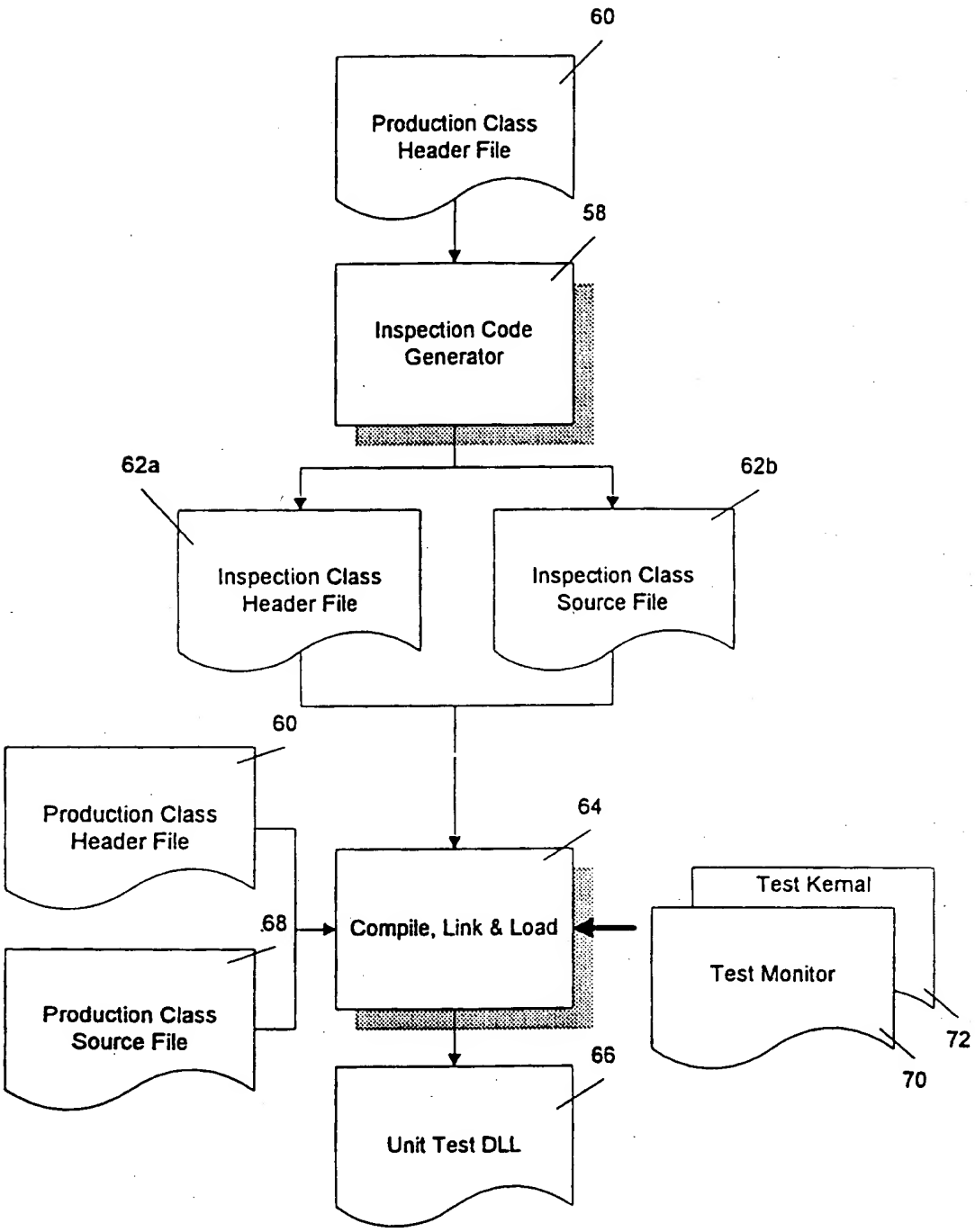

60
Production Class Header File
58
Inspection Code Generator
62a
62b
Inspection Class Header File
Inspection Class Source File
60
Production Class Header File
64
Test Kernal
68
Compile, Link & Load
Test Monitor
Production Class Source File
72
66
70
Unit Test DLL

Figure 9

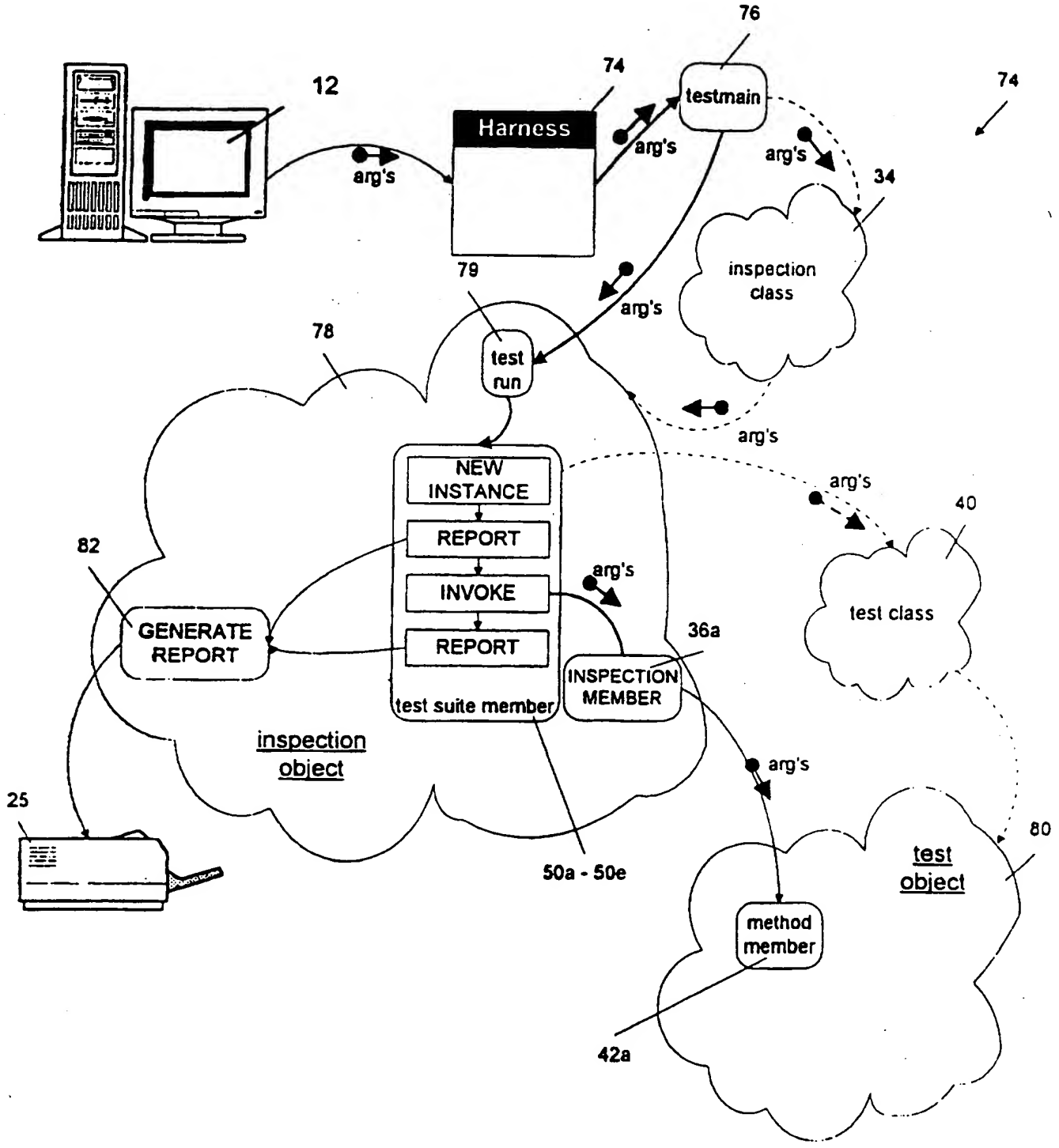

76
74
12
Harness
testmain
arg's
arg's
arg's
74
34
inspection class
79
arg's
78
test run
arg's
NEW INSTANCE
REPORT
INVOKE
REPORT
arg's
40
82
GENERATE REPORT
test class
36a
INSPECTION MEMBER
test suite member
inspection object
arg's
25
80
test object
50a - 50e
method member
42a

Figure 10

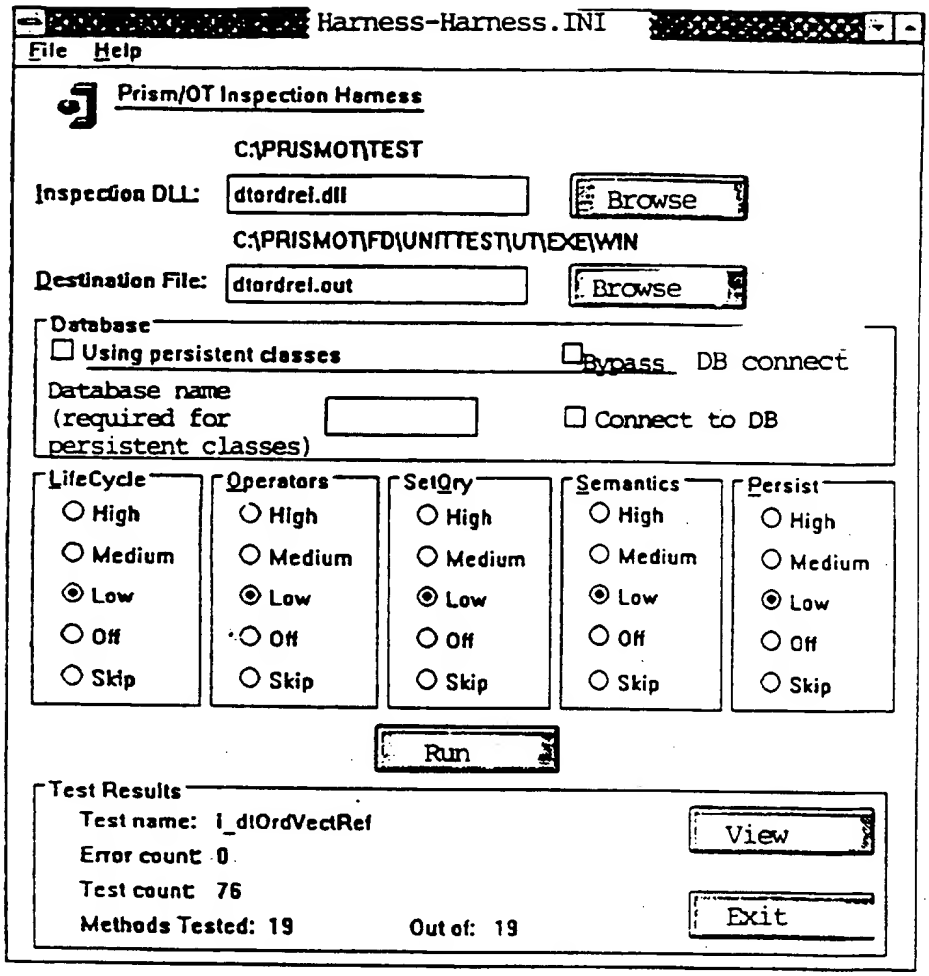

Harness-Harness.INI
File Help
Prism/OT Inspection Harness
C:\PRISMOT\TEST
Inspection DLL:
dtordrel.dll
Browse
C:\PRISMOT\FD\UNITTEST\UT\EXE\WIN
Destination File:
dtordrel.out
Browse
Database
Using persistent classes
Bypass DB connect
Database name (required for persistent classes)
Connect to DB
LifeCycle
Operators
SetOry
Semantics
Persist
High
Medium
Low
Off
Skip
Run
Test Results
Test name: I_dtOrdVectRef
Error count: 0
Test count: 76
Methods Tested: 19
Out of: 19
View
Exit

Figure 11

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US95/06059

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) :G06F 9/445,9/45
US CL :395/700

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 395/700, 364/275.5

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

IEEE online

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	Meyer, Object-Oriented Software Construction, Prentise-Hall, 1988, pages 65-104, and especially pages 347-348.	1-80
X	US, A, 5,093,914 (Coplien, et al.) 03 March 1992, col. 1-26.	1-80
Y	US, A, 4,885,717 (Beck, et al.), 05 December 1989, col. 1-16.	1-80
Y	US, A, 4,989,132 (Mellander, et al.), 29 January 1991, col. 46-54.	1-80

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be part of particular relevance

"E" earlier document published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
04 SEPTEMBER 1995	22 SEP 1995
Name and mailing address of the ISA/US Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231 Facsimile No. (703) 305-3230	Authorized officer JON BACKENSTOSE Telephone No. (703) 305-9661

Form PCT/ISA/210 (second sheet)(July 1992)*